Global Call API for Host Media Processing on Windows

Programming Guide

April 2005

05-2409-001

Publication Date: April 2005

Document Number: 05-2409-001

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

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

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

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

Contents

Figures

Tables

Revision History

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

Document No.	Publication Date	Description of Revisions
05-2409-001	April 2005	Initial version of document. Much of the information contained in this document was previously published in the *Global Call API for Windows Operating Systems Programming Guide*, document number 05-1867-002.

About This Publication

The following topics provide information about this publication:

Purpose

This publication provides guidelines for using the Global Call API to build computer telephony applications that require call control functionality. Such applications include, but are not limited to, Call Routing, Enhanced Services, Unified Messaging, Voice Messaging, LAN Telephony Services, Computer Telephony Services, Switching, PBX, Interactive Voice Response, Help Desk and Work Flow applications. This publication is a companion guide to the *Global Call API Library Reference* that provides details on the functions and parameters in the Global Call API library and the *Global Call IP Technology Guide* that provides IP-specific information.

Host Media Processing (HMP) software performs media processing tasks on general-purpose servers based on Intel® architecture without the need for specialized hardware. When installed on a system, HMP performs like a virtual DM3 board to the customer application, but all media processing takes place on the host processor. In this document, the term “board” represents the virtual DM3 board.

Applicability

This document is published for Intel® NetStructure™ Host Media Processing Software.

Intended Audience

This publication is written for the following audience:

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

How to Use This Publication

Refer to this publication after you have installed the hardware and the system software which includes the Global Call software.

This publication assumes that you are familiar with the Windows operating system and the C programming language.

The information in this guide is organized as follows:

- Chapter 1, "Product Description" provides an overview of the Global Call development software.
- Chapter 2, "Programming Models" describes the supported programming models in the Windows environment.
- Chapter 3, "Call State Models" describes the call state models used by Global Call.
- Chapter 4, "Event Handling" describes how to handle Global Call events.
- Chapter 6, "Error Handling" describes the error handling facilities provided by Global Call.
- Chapter 5, "Application Development Guidelines" provides guidelines when developing applications that use Global Call.
- Chapter 7, "Call Control" describes basic call control capabilities, resource routing and feature extension features and provided by Global Call.
- Chapter 8, "Alarm Handling" describes how Global Call can be used to handle alarms.
- Chapter 9, "Real Time Configuration Management" describes how Global Call can be used for real time configuration of parameters associated with the interface.
- Chapter 10, "Handling Service Requests" describes the generic service request facility provided by Global Call.
- Chapter 11, "Using Global Call to Implement Call Transfer" provides general information on the implementation of unsupervised (blind) and supervised call transfer.
- Chapter 12, "Building Applications" provides guidelines for building applications that use the Global Call software.
- Chapter 13, "Debugging" provides pointers to where technology-specific debugging information can be obtained.
- The Glossary provides a definition of terms used in this guide.

Related Information

Refer to the following sources for more information:

- *Global Call API Library Reference*
- *Global Call IP Technology Guide*
- *Standard Runtime Library API Programming Guide*.
- *Standard Runtime Library API Library Reference*.

- The Release Update for your HMP software, which may include updates to this manual, available on the Telecom Support Resources website at: *http://www.intel.com/design/network/products/telecom/software/index.htm*
- *http://developer.intel.com/design/telecom/support/* (for technical support)
- *http://www.intel.com/design/network/products/telecom* (for product information)

Product Description 1

This chapter describes the Global Call software. Topics include:

1.1 Global Call Software Overview

Global Call development software provides a common signaling interface for network-enabled applications, regardless of the signaling protocol needed to connect to the local telephone network. The signaling interface provided by Global Call software facilitates the exchange of call control messages between the telephone network and virtually any network-enabled application. Global Call software enables developers to create applications that can work with signaling systems worldwide, regardless of the network to which the applications are connected. Global Call software is ideal for high-density, network-enabled solutions, such as voice, data, and video applications, where the supported hardware and signaling technology can vary widely from country to country.

As an example, the signal acknowledgement or information flow required to establish a call may vary from country to country. Rather than requiring the application to handle low-level details, Global Call software offers a consistent, high-level interface to the user and handles each country's unique protocol requirements transparently to the application.

The Global Call software comprises three major components:

Global Call Application Programming Interface (API)
: A common, extensible API providing network interfaces to higher levels of software. Application developers use API function calls in their computer telephony applications. The Global Call API is the preferred call control interface.

Call Control Libraries
: A set of libraries that provide the interface between the Global Call API and the various network signaling protocols.

Global Call Protocols
: Network signaling protocols, such as IP H.323 and SIP can be invoked by the Global Call API to facilitate call control.

1.2 Global Call Feature Categories

The Global Call development software provides many features allowing for the development of flexible and robust applications. The features fall into one of two categories:

- Call Control Features
- Operation, Administration and Maintenance Features

1.2.1 Call Control Features

The Global Call development software provides the following call control features:

Basic Call Control
: Includes basic call control features such as the ability to make a call, detect a call, answer a call, release a call, etc. The implementation of these capabilities is based on the basic call state model, which is a common model for all network technologies. See Section 3.2, "Basic Call Model" for more information on the basic call model.

Feature Transparency and Extension (FTE)
: Provides the ability to extend the capabilities of Global Call to handle features that are specific to a particular technology so that those features are accessible via the Global Call interface. See Section 7.3, "Feature Transparency and Extension" for more information.

1.2.2 Operation, Administration and Maintenance Features

The Global Call development software provides the following features that facilitate the operation, administration and maintenance of Global Call applications:

Error Handling Functionality
: When an error occurs, Global Call provides functions that enable an application to retrieve more information about the error. See Chapter 6, "Error Handling" for more information.

Event Handling Functionality
: Provides the ability to handle and process events, including the ability to disable and enable events and to retrieve event information. See Chapter 4, "Event Handling" for more information.

Global Call Alarm Management System (GCAMS)
: Provides the ability to manage alarms. GCAMS provides Global Call applications with the ability to receive extensive alarm information that can be used to troubleshoot line problems. See Chapter 8, "Alarm Handling" for more information.

Real Time Configuration Management (RTCM)
: Allows the modification of call control and protocol elements in real time, providing a single common user interface for configuration management. See Chapter 9, "Real Time Configuration Management" for more information.

Global Call Service Request (GCSR)
: Enables an application to send a request for a service to a remote device. Examples of the types of services that this feature supports are device registration, channel setup, call setup, information requests, or other kinds of requests that need to be made between two devices across the network. See Chapter 10, "Handling Service Requests" for more information.

Library Information Functions

Enables an application to get information about the call control libraries being used. See the *Global Call API Library Reference* for more information about these functions.

Debugging Facilities

Global Call provides powerful debugging capabilities for troubleshooting protocol-related problems, including the ability to generate a detailed log file. See the *Global Call IP Technology Guide* for information on the debugging facilities available when using Global Call with IP technology.

1.3 Global Call Architecture

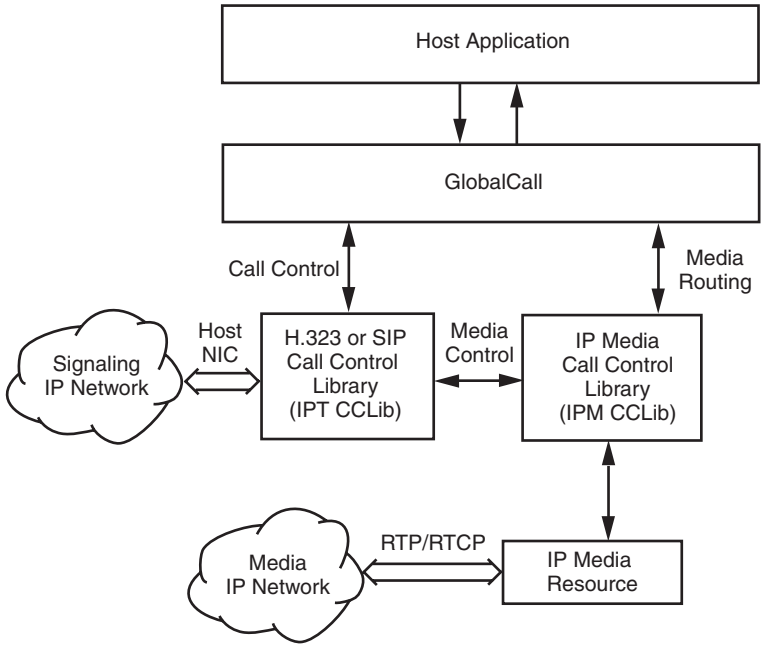
The Global Call development software architecture is based on the Intel® Dialogic® architecture that supports Host Media Processing (HMP) software. The architecture is described in the following topics:

- Overview for IP Technology
- Global Call API

1.3.1 Overview for IP Technology

Figure 1 shows a system-level view of the Global Call architecture for IP technology.

Figure 1. Global Call Architecture for IP Technology



1.3.2 Global Call API

The Global Call API is a call control API. Similar to other Intel Dialogic APIs (such as the Voice API), the Global Call API uses the Standard Runtime Library (SRL) API to deliver response events to its API commands. The Global Call API and other Intel Dialogic APIs form a family of APIs that use the underlying services provided by the SRL API.

The Global Call API provides a collection of functions supporting call control operations as well as functions to support operation, administration and maintenance tasks. See the *Global Call API Library Reference* for detailed information about each function.

1.4 Call Control Libraries

Each supported network technology requires a call control library to provide the interface between the network and the Global Call library. The call control libraries currently supported by the Global Call API for HMP are as follows:

GC_CUSTOM1_LIB
: The first of two call control library place holders for custom call control libraries. Any third-party Global Call compatible call control library can be used as a custom library. The Global Call library supports up to two custom libraries.

GC_CUSTOM2_LIB
: The second of two call control library place holders for custom call control libraries. Any third-party Global Call compatible call control library can be used as a custom library. The Global Call library supports up to two custom libraries.

GC_H3R_LIB
: The call control library that controls access to IP network interfaces. This call control library supports IP H.323 and SIP protocols.

GC_IPM_LIB
: The call control library that provides access to IP media resources.

1.4.1 Starting Call Control Libraries

Call control libraries must be started before they can be used by the Global Call functions. The call control libraries are started when a **gc_Start( )** function is issued. The **gc_Start( )** function allows the selective starting of call control libraries where the application can specify if all the call control libraries are to be started or only specified libraries are to be started. The application can also start a custom call control library that is not supported by Global Call. See the *Global Call API Library Reference* for more information about the **gc_Start( )** function.

1.4.2 Call Control Library States

The initial state of all the call control libraries is the Configured state. When a call control library is successfully started, the library will be in the Available state. If the call control library fails to start, the library will be in the Failed state as shown in the diagram below. If the call control library is not started, it will remain in the Configured state.

Figure 2. Call Control Library States

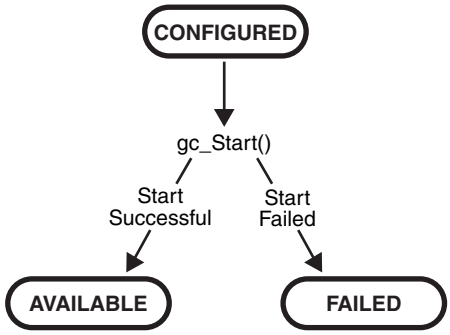


Table 1 describes the different states of a call control library.

Table 1. Call Control Library States

State	Description
Configured	A library that is supported by Global Call is considered a configured library.
Available	A library that has been successfully started is considered to be available for use by a Global Call application.
Failed	A library that has failed to start is considered to be unavailable for use by a Global Call application.

Each configured call control library is assigned an ID number by Global Call. Each library also has a name in an ASCII string format. Library functions perform tasks such as converting a call control library ID to an ASCII name and vice-versa, determining the configured libraries, determining the available libraries, determining the libraries started and the libraries that failed to start, and other library functions.

The following functions are the call control library information functions. All the library functions are synchronous, thus they return without a termination event.

- **gc_CCLibIDToName( )**
- **gc_CCLibNameToID( )**
- **gc_CCLibStatusEx( )**
- **gc_GetVer( )**

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

1.5 Global Call Object Identifiers

The Global Call API is call-oriented, that is, each call initiated by the application or network is assigned a Call Reference Number (CRN) for call control and tracking purposes. Call handling is independent of the line device over which the call is routed. Each line device or device group is assigned a Line Device Identifier (LDID) that enables the application to address any resource or

group of resources using a single device identifier. Certain features, such as Feature Transparency and Extension (FTE), Real Time Configuration Management (RTCM), and Global Call Service Request (GCSR), operate on a basic entity called a Global Call target object. Target objects are identified by a target type and a target ID.

The following topics provide more detailed information:

- Line Device Identifier
- Call Reference Number
- Object Identifiers and Resource Sharing Across Processes
- Target Objects

1.5.1 Line Device Identifier

A Line Device Identifier (LDID) is a unique logical number assigned to a specific resource (for example, a time slot) or a group of resources within a process by the Global Call library. Minimally, the LDID number will represent a network resource. For example, both a network resource and a voice resource are needed to process an R2 MFC dialing function. Using Global Call, a single LDID number is used by the application (or thread) to represent this combination of resources for call control.

An LDID number is assigned to represent a physical device(s) or logical device(s) that will handle a call, such as a network interface resource, when the **gc_OpenEx( )** function is called. This identification number assignment remains valid until the **gc_Close( )** function is called to close the line device.

When an event arrives, the application (or thread) can retrieve the LDID number associated with the event by using the linedev field of the associated METAEVENT structure. The LDID is retrieved using the **gc_GetMetaEvent( )** function.The LDID is retrieved using the **gc_GetMetaEvent( )** or the **gc_GetMetaEventEx( )** function.

1.5.2 Call Reference Number

A Call Reference Number (CRN) is a means of identifying a call on a specific line device. A CRN is created by the Global Call library when a call is requested by the application, thread or network.

With the CRN approach, the application (or thread) can access and control the call without any reference to a specific physical port or line device. CRNs are assigned to both inbound and outbound calls:

Inbound calls
: The CRN is assigned via the **gc_WaitCall( )** function. For more information on **gc_WaitCall( )**, see the *Global Call API Library Reference*.

Outbound calls
: The CRN is assigned via either the **gc_MakeCall( )** function. For more information on these functions, see the *Global Call API Library Reference*.

This CRN has a single LDID associated with it, for example, the line device on which the call was made. However, a single line device may have multiple CRNs associated with it (that is, more than one call may exist on a given line). A line device can have a maximum of 20 CRNs associated with it. At any given instant, each CRN is a unique number within a process. After a call is terminated and the **gc_ReleaseCallEx( )** function is called to release the resources used for the call, the CRN is no longer valid.

1.5.3 Object Identifiers and Resource Sharing Across Processes

The CRNs and LDIDs assigned by the Global Call API library can **not** be shared among multiple processes. These assigned CRNs and LDIDs remain valid only within the process invoked. That is, for call control purposes, you should not open the same physical device from more than one process, nor from multiple threads in a Windows environment. If either of these conditions exist, unpredictable results may occur.

1.5.4 Target Objects

A target object provides a way of identifying a particular entity that is maintained by a specific software module. In API function calls, the target object is specified by a pair of parameters, the **target_type** and **target_ID**:

target_type
: Identifies the kind of software module and the entity that it maintains. For example, the target type GCTGT_GCLIB_CHAN represents the Global Call Library and a channel entity that it maintains.

target_ID
: Identifies the specific target object, such as a line device ID (LDID), which is generated by Global Call at runtime.

Table 2 shows the combinations of physical or logical entities and software module entities that can make up a target type (**target_type**).

Table 2. Supported Target Types

Software Module	Entity			
	System	Network Interface	Channel	CRN
GCLib	S	S	S	S
CCLib	S	S	S	S
Protocol	SV	SV	SV	
Firmware		SV	SV	

S = Supported
SV = Supported with Variances, see the *Global Call IP Technology Guide* for more information.

The possible software modules include:

- GCLib
- CCLib

- Protocol
- Firmware

The possible entities include:

System
NIC

Network Interface
logical board or virtual board

Channel
time slot

CRN
call reference number

A target type (**target_type**) name is composed of the prefix, GCTGT, which stands for Global Call Target, a software module name, such as GCLIB, and an entity name, such as NETIF. For example, the target type GCTGT_GCLIB_NETIF, indicates that the desired target type is a network interface maintained by the Global Call library.

A target ID (**target_ID**) identifies the specific object that is located within the category defined by the target type (**target_type**). A target ID can be any of the following:

- line device ID (LDID)
- call reference number (CRN)
- Global Call library ID (GCGV_LIB)
- call control library ID (CCLib ID)
- protocol ID

The types and IDs for target objects are defined at the Global Call level. Table 3 shows the target types, as described in Table 2, with various target IDs to represent valid target objects.

Table 3. Target Types and Target IDs

Target Type	Target ID	Description
GCTGT_GCLIB_SYSTEM †	GCGV_LIB	Global Call library module target object.
GCTGT_GCLIB_NETIF	Global Call Line device ID	Network interface target object in Global Call Library module.
GCTGT_CCLIB_NETIF	Global Call Line device ID	Network interface target object in call control library module.
GCTGT_GCLIB_CHAN	Global Call Line device ID	Channel target object in Global Call library module.
GCTGT_CCLIB_CHAN	Global Call Line device ID	Channel target object in call control library module.
† Target types that can only be used by functions issued in **synchronous** mode. If a function uses one of these target types in asynchronous mode, an error will be generated. The functions that can use these target types are **gc_SetConfigData( )**, **gc_ReqService( )**, and **gc_RespService( )**.		

Table 3. Target Types and Target IDs

Target Type	Target ID	Description
GCTGT_GCLIB_CRN	Global Call CRN	CRN target object in Global Call library module.
GCTGT_CCLIB_CRN	Global Call CRN	CRN target object in call control library module.
† Target types that can only be used by functions issued in **synchronous** mode. If a function uses one of these target types in asynchronous mode, an error will be generated. The functions that can use these target types are **gc_SetConfigData( )**, **gc_ReqService( )**, and **gc_RespService( )**.		

Target Object Availability

Except for the GCTGT_GCLIB_SYSTEM target object, all target IDs are generated or assigned by the Global Call API when the target object is created (for physical targets) or loaded (for software targets). Table 4 shows when a target object becomes available and when it becomes unavailable, depending on the target type.

Table 4. Target Object Availability

Target Type	Target Object Available	Target Object Unavailable
GCTGT_GCLIB_SYSTEM	After **gc_Start( )**	After **gc_Stop( )**
GCTGT_GCLIB_CRN GCTGT_CCLIB_CRN	After a call is created (**gc_MakeCall( )** returns or GCEV_OFFERED is received)	After **gc_ReleaseCallEx( )**
GCTGT_GCLIB_NETIF GCTGT_CCLIB_NETIF GCTGT_GCLIB_CHAN GCTGT_CCLIB_CHAN	After **gc_OpenEx( )**	After **gc_Close( )**

Retrieving Target IDs

Before the Global Call application can retrieve, update, or query the configuration data of a target object, it should obtain the target ID as shown in Table 5.

Table 5. Obtaining Target IDs

Target ID	Procedure for Obtaining Target ID
GCGV_LIB	After the call control library has been successfully started (that is, after the **gc_Start( )** function is called), the target object's CCLib ID can be obtained by calling the **gc_CCLibNameToID( )** function.
Global Call CRN	After a call target object is created, its target object ID (that is, the Global Call CRN) will be an output of the **gc_MakeCall( )** function or provided by the metaevent associated with the GCEV_OFFERED event.

Programming Models 2

This chapter describes the programming models supported by Global Call. Topics include:

2.1 Programming Models Overview

The Global Call development software supports application development using asynchronous programming models. By usage, the asynchronous models are often said to use asynchronous **mode**. Asynchronous mode programming is introduced briefly in this chapter and described in more detail in the *Standard Runtime Library API Programming Guide*.

2.2 Asynchronous Mode Programming

Programming in asynchronous mode in Windows is described in the following topics:

- Asynchronous Model Overview
- Asynchronous Model with Event Handlers
- Asynchronous with Windows Callback Model
- Asynchronous with Win32 Synchronization Model
- Extended Asynchronous Programming Model

2.2.1 Asynchronous Model Overview

Asynchronous mode programming is characterized by the calling thread performing other processing while a function executes. At completion, the application receives event notification from the SRL and then the thread continues processing the call on a particular channel.

A function called in the asynchronous mode returns control immediately after the request is passed to the device driver and allows thread processing to continue. A termination event is returned when the requested operation completes, thus allowing the Intel Dialogic operation (state machine processing) to continue.

Caution: In general, when a function is called in asynchronous mode, and an associated termination event exists, the **gc_Close( )** function should not be called until the termination event has been received. In order to disable **gc_WaitCall( )**, **gc_ResetLineDev( )** should be called. If this is not done, there are potential race conditions under which the application may crash with a segmentation fault.

Functions may be initiated asynchronously from a single thread and/or the completion (termination) event can be picked up by the same or a different thread that calls the **sr_waitevt( )**

and **gc_GetMetaEvent( )** functions. When these functions return with an event, the event information is stored in the METAEVENT data structure. The event information retrieved determines the exact event that occurred and is valid until the **sr_waitevt( )** and **gc_GetMetaEvent( )** functions are called again.

For Windows environments, the asynchronous models provided for application development also include:

- asynchronous model with event handlers
- asynchronous with Windows callback
- asynchronous with Win32 synchronization
- extended asynchronous programming

The asynchronous programming models are recommended for more complex applications that require coordinating multiple tasks. Asynchronous model applications typically run faster than synchronous models and require lower levels of system resources. Asynchronous models reduce processor loading because of the reduced number of threads inherent in asynchronous models and the elimination of scheduling overhead. Asynchronous models use processor resources more efficiently because multiple channels are handled in a single thread or in a few threads. See Section 5.1, “General Programming Tips”, on page 63 for details. Of the asynchronous models, the asynchronous with SRL callback model and the asynchronous with Windows callback model provide the tightest integration with the Windows message/event mechanism. Asynchronous model applications are typically more complex than corresponding synchronous model applications due to a higher level of resource management (that is, the number of channels managed by a thread and the tracking of completion events) and the development of a state machine.

After the application issues an asynchronous function, the application uses the **sr_waitevt( )** function to wait for events on Intel Dialogic devices. All event coding can be accomplished using switch statements in the main thread. When an event is available, event information may be retrieved using the **gc_GetMetaEvent( )** function. Retrieved event information is valid until the **sr_waitevt( )** function is called again. The asynchronous model does not use event handlers to process events.

In this model, the SRL handler thread must be initiated by the application by setting the SR_MODELTYPE value to SR_STASYNC.

2.2.2 Asynchronous Model with Event Handlers

The asynchronous with event handlers model uses the **sr_enbhdlr( )** function to automatically create the SRL handler thread. The application does not need to call the **sr_waitevt( )** function since the thread created by the **sr_enbhdlr( )** already calls the **sr_waitevt( )** function to get events. Each call to the **sr_enbhdlr( )** function allows the Intel Dialogic events to be serviced when the operating system schedules the SRL handler thread for execution.

Note: The SR_MODELTYPE value must **not** be set to SR_STASYNC because the SRL handler thread must be created by the **sr_enbhdlr( )** call. The event handler must **not** call the **sr_waitevt( )** function or any synchronous Intel Dialogic function.

Individual handlers can be written to handle events for each channel. The SRL handler thread can be used when porting applications developed for other operating systems.

2.2.3 Asynchronous with Windows Callback Model

The asynchronous with Windows callback model allows an asynchronous application to receive SRL event notification through the standard Windows message handling scheme. This model is used to achieve the tightest possible integration with the Windows messaging scheme. Using this model, the entire Intel Dialogic portion of the application could be run on a single thread. This model calls the **sr_NotifyEvt( )** function once to define a user-specified application window handle and a user-specified message type. When an event is detected, a message is sent to the application window. The application responds by calling the **sr_waitevt( )** function with a 0 **timeout** value. For Global Call events and optionally for non-Global Call events, the application **must** then call the **gc_GetMetaEvent( )** function before servicing the event.

In this model, the SRL event handler thread must be initiated by the application by setting the SR_MODELTYPE value to SR_STASYNC. For detailed information on this programming model, see the *Standard Runtime Library API Programming Guide*.

2.2.4 Asynchronous with Win32 Synchronization Model

The asynchronous with Win32 synchronization model allows an asynchronous application to receive SRL event notification through standard Windows synchronization mechanisms. This model uses one thread to run all Intel Dialogic devices and thus requires a lower level of system resources than the synchronous model. This model allows for greater scalability in growing systems. For detailed information on this programming model, see the *Standard Runtime Library API Programming Guide*.

2.2.5 Extended Asynchronous Programming Model

The extended asynchronous programming model is basically the same as the asynchronous model except that the application uses multiple asynchronous threads, each of which controls multiple devices. In this model, each thread has its own specific state machine for the devices that it controls. Thus, a single thread can look for separate events for more than one group of channels. This model may be useful, for example, when you have one group of devices that provides fax services and another group that provides interactive voice response (IVR) services, while both groups share the same process space and database resources. The extended asynchronous model can be used when an application needs to wait for events from more than one group of devices and requires a state machine.

Because the extended asynchronous model uses only a few threads for all Intel Dialogic devices, it requires a lower level of system resources than the synchronous model. This model also enables using only a few threads to run the entire Intel Dialogic portion of the application.

Whereas default asynchronous programming uses the **sr_waitevt( )** function to wait for events specific to one device, extended asynchronous programming uses the **sr_waitevtEx( )** function to wait for events specific to a number of devices (channels).

Note: Do not use the **sr_waitevtEx( )** function in combination with either the **sr_waitevt( )** function or event handlers.

This model can run an entire application using only a few threads. When an event is available, the **gc_GetMetaEventEx( )** function must be used to retrieve event-specific information. The values returned are valid until the **sr_waitevtEx( )** function is called again. Event commands can be executed from the main thread through switch statements; the events are processed immediately.

The extended asynchronous model calls the **sr_waitevtEx( )** function for a group of devices (channels) and polls for (waits for) events specific to that group of devices. In this model, the SRL event handler thread is **not** created (the SR_MODELTYPE value is set to SR_STASYNC) and the **sr_enbhdlr( )** function in **not** used.

In the extended asynchronous model, functions are initiated asynchronously from different threads. A thread waits for events using the **sr_waitevtEx( )** function. The event information can be retrieved using the **gc_GetMetaEventEx( )** function. When this function returns, the event information is stored in the METAEVENT data structure.

Caution: When calling the **gc_GetMetaEventEx( )** function from multiple threads, ensure that your application uses unique thread-related METAEVENT data structures (thread local variables or local variables), or ensure that the METAEVENT data structure is not overwritten until all processing of the current event has completed.

The event information retrieved determines the exact event that occurred and is valid until the **sr_waitevtEx( )** function returns with another event.

Call State Models 3

This chapter describes the call state models provided by Global Call. Topics include the following:

3.1 Call State Model Overview

Global Call maintains a generic call model from which technology-specific call models can be derived. Some technologies support only a subset of the complete call model. The call establishment and termination procedures are based on this call model. The following sections describe the call states associated with the basic call model and configuration options.

3.2 Basic Call Model

Each call received or generated by Global Call is processed through a series of states, where each state represents the completion of certain tasks or the current status of the call. Some states in the basic call model are optional and can be enabled or disabled selectively. Only the optional states can be enabled or disabled. Every technology or call control library has a default call state model consisting of all the states it can possibly support from the basic call model. If a state is disabled, all corresponding events are disabled. If a state is enabled, all corresponding events are enabled.

The call states change in accordance with the sequence of functions called by the application and the events that originate in the network and system hardware. The current state of a call can be changed by:

- Function call returns
- Termination events (indications of function completion)
- Unsolicited events

The states of the basic call model are described in the following sections:

- Basic Call States at the Inbound Interface
- Basic Call States at the Outbound Interface
- Basic Call States for Call Termination

3.2.1 Basic Call States at the Inbound Interface

The basic inbound call states are as follows:

Null state (GCST_NULL)
: This state indicates that no call is assigned to the channel (time slot or line). This is the initial state of a channel when it is first opened. This state is also reached when a call is released or after the channel is reset. A channel in this state is available for inbound calls after being initialized to receive incoming calls.

Call Detected (GCST_DETECTED)
: An incoming call has been received but not yet offered to the application. In this state, the call is being processed, which typically involves waiting for more information or allocating a resource. Although the call is not yet offered to the application, this state is for informational purposes to reduce glare conditions since the application is aware of the presence of a call on the channel.

Call Offered (GCST_OFFERED)
: This state exists for an incoming call when the user application has received a call establishment request but has not yet responded. The newly arrived inbound call is offered to the user application to be accepted, answered, rejected, etc. Call information is typically available at this time to be examined so that the application can determine the appropriate action to take with regards to the call.

Call Routing (GCST_CALLROUTING)
: This state exists for an incoming call when the user has sent an acknowledgement that all call information necessary to effect call establishment has been received. The acknowledgement can be sent from the Offered or the GetMoreInfo state if all the information has been received. This transition typically involves the sending of Call Routing tones or technology specific messages. The application can now accept or answer the call. (This state is optional and may not be supported in all technologies. See the *Global Call IP Technology Guide* for information.)

Call Accept (GCST_ACCEPT)
: This state indicates that the incoming call was offered and accepted by the application. The user on the inbound side has indicated to the calling party that the destination user is alerting or ringing but has not yet answered.

Call Connected (GCST_CONNECTED)
: This is a common state that exists for an incoming call when the user has answered the call.

3.2.2 Basic Call States at the Outbound Interface

The basic outbound call states are as follows:

Null state (GCST_NULL)
: This state indicates that no call is assigned to the channel (time slot or line). This is the initial state of a channel when it is first opened. This state is also reached when a call is released or after the channel is reset. The channel in this state is available for making outbound calls.

Call Dialing (GCST_DIALING)
: This state exists for an outgoing call when an outbound call request is made. The call signaling or message is in the process of being prepared for transfer or being transferred across the

telephony network. In response, the remote side may acknowledge the call, accept the call or answer the call.

Call Proceeding (GCST_PROCEEDING)
: This state exists for an outgoing call when the user has received an acknowledgement that all call information necessary to effect call establishment has been received and the call is proceeding. The remote side can now accept or answer the call. (This state is optional and may not be supported in all technologies. See the *Global Call IP Technology Guide* for information.)

Call Alerting (GCST_ALERTING)
: This state exists for an outgoing call when the calling user has received an indication that remote user alerting has been initiated, typically ringing. The outbound call has been delivered to the remote party, which has not yet answered the call.

Call Connected (GCST_CONNECTED)
: This is a common state that exists for an outgoing call when the user has received an indication that the remote user has answered the call. The calling and called parties are connected and the call is therefore active on the related call channel.

3.2.3 Basic Call States for Call Termination

The basic call termination states are as follows:

Call Disconnected (GCST_DISCONNECTED)
: This state indicates that the remote party has disconnected the call. The remote party can disconnect the call prior to establishing a connection, that is, while the call setup is in progress. Thus, the call does not have to be in the connected state before it can be disconnected. The user must respond by dropping the call and releasing the internal resources allocated for the call.

Call Idle (GCST_IDLE)
: This state indicates that the local user has dropped the call. This may be a termination initiated by the local user or a response to the remote side disconnecting the call. While the call no longer exists, internal system resources committed to servicing the call are still present. The user must release these resources, as they are no longer required.

3.3 Basic Call Model Configuration Options

Depending on the specific technology, the following options are available for configuring the technology call control layer or the application:

Call State
: If a state is disabled, the corresponding call state event is also disabled.

Call State Event
: Call state transition events are masked so that the events are not generated.

Call Acknowledgement
: An acknowledgement is sent to indicate to the remote side that the call has been received but more information is required to proceed with the call.

Call Proceeding
: Call proceeding information is sent to the remote side when an incoming call is received and all the information required to proceed with the call is available.

Minimum Information
: A minimum amount of destination address information, such as DNIS, is collected before the call is offered to the application.

3.3.1 Call State Configuration

Some states in the basic call model are optional and can be enabled or disabled selectively. Every technology or call control library has a default call state model consisting of all the states it can possibly support from the basic call model. If a state is disabled, the corresponding call state event will also be disabled. If a state is enabled, the event mask setting still determines which call state events are sent to the application.

This configuration can be done by issuing the **gc_SetConfigData( )** function with a **target_type** of GCTGT_GCLIB_CHAN and a **target_ID** of a line device, and passing the appropriate set ID and parameter IDs. The set ID used in this context is GCSET_CALLSTATE_MSK and the relevant parameter IDs are:

GCACT_ADDMSK
: Enable the call states specified in the value in addition to other states already enabled.

GCACT_SUBMSK
: Disable all the call states specified in the value.

GCACT_SETMSK
: Enable the call states specified in the value and disable other optional states that are already enabled.

The GCACT_ADDMSK, GCACT_SUBMSK and GCACT_SETMSK parameter IDs can be assigned one of the following values (of type GC_VALUE_LONG), or an ORed combination of the values:

- GCMSK_ALERTING_STATE
- GCMSK_DETECTED_STATE
- GCMSK_PROCEEDING_STATE

See the *Global Call API Library Reference* for more information on the **gc_SetConfigData( )** function.

3.3.2 Call State Event Configuration

Some call state transition events can be masked so that the events are not generated. Although an event may be masked, the corresponding call state transition can still take place. This configuration can be done by issuing the **gc_SetConfigData( )** function with a **target_type** of GCTGT_GCLIB_CHAN and a **target_ID** of a line device, and passing the appropriate set ID and parm IDs.

The set ID used in this context is GCSET_CALLEVENT_MSK and the relevant parm IDs are:

GCACT_ADDMSK
: Enable the notification of events specified in the value in addition to previously enabled events.

GCACT_SUBMSK
: Disable notification of the events specified in the value.

GCACT_SETMSK
: Enable the notification of events specified in the value and disables notification of any event not specified.

The GCACT_ADDMSK, GCACT_SUBMSK and GCACT_SETMSK parm IDs can be assigned one of the following values (of type GC_VALUE_LONG), or an ORed combination of the values:

- GCMSK_ALERTING
- GCMSK_DETECTED
- GCMSK_DIALING
- GCMSK_PROCEEDING

Note: Using the **gc_SetConfigData( )** function with a **target_ID** of a board device to mask events for all devices associated with a board is **not** supported. Call state events can be masked on a per line device basis only.

See the *Global Call API Library Reference* for more information on the **gc_SetConfigData( )** function.

3.3.3 Call Proceeding Configuration

When an incoming call is received and all the information required to proceed with the call is available, an indication that the call is proceeding is usually sent to the remote side for informational purposes. Either the technology call control layer or the application can be configured to send a call proceeding indication to the remote side. This can be done by issuing the **gc_SetConfigData( )** function. The set ID used in this context is GCSET_CALL_CONFIG and the relevant parm ID is:

GCPARM_CALLPROC
: Specify whether call proceeding indication is provided by the application or the technology call control layer.

The GCPARM_CALLPROC parm ID can be assigned one of the following values (of type GC_VALUE_INT):

- GCCONTROL_APP (application controlled)
- GCCONTROL_TCCL (technology call control layer controlled)

See the *Global Call API Library Reference* for more information on the **gc_SetConfigData( )** function.

3.4 Basic Call Control in Asynchronous Mode

This section describes and illustrates the basic call model and state transitions for call control in asynchronous mode. This section also describes the process for call establishment for both inbound and outbound calls and call termination in the asynchronous mode.

The procedures for establishing and terminating calls in the asynchronous mode are described in the following sections:

- Inbound Calls in Asynchronous Mode
- Outbound Calls in Asynchronous Mode
- Call Termination in Asynchronous Mode

Caution: In general, when a function is called in asynchronous mode, and an associated termination event exists, the **gc_Close( )** function should not be called until the termination event has been received. Otherwise, the behavior is undefined.

3.4.1 Inbound Calls in Asynchronous Mode

This section describes how calls are established and shows call scenarios for asynchronous inbound calls. The following topics describe the processing of inbound calls in asynchronous mode:

- Inbound Calls in Asynchronous Mode Overview
- Channel Initialization
- Call Detection
- Call Offered
- Call Routing
- Call Acceptance
- Call Establishment
- Call Failure
- Abandoned Calls
- Inbound Call Scenarios in Asynchronous Mode

3.4.1.1 Inbound Calls in Asynchronous Mode Overview

Figure 3 illustrates a Basic Inbound Call Model, which shows the call states associated with establishing a call in asynchronous mode. All calls start from a Null state. The call establishment process for inbound calls is shown in Figure 3.

See Table 6, “Asynchronous Inbound Call State Transitions”, on page 38 for a summary of the call state transitions.

Figure 3. Basic Asynchronous Inbound Call State Diagram

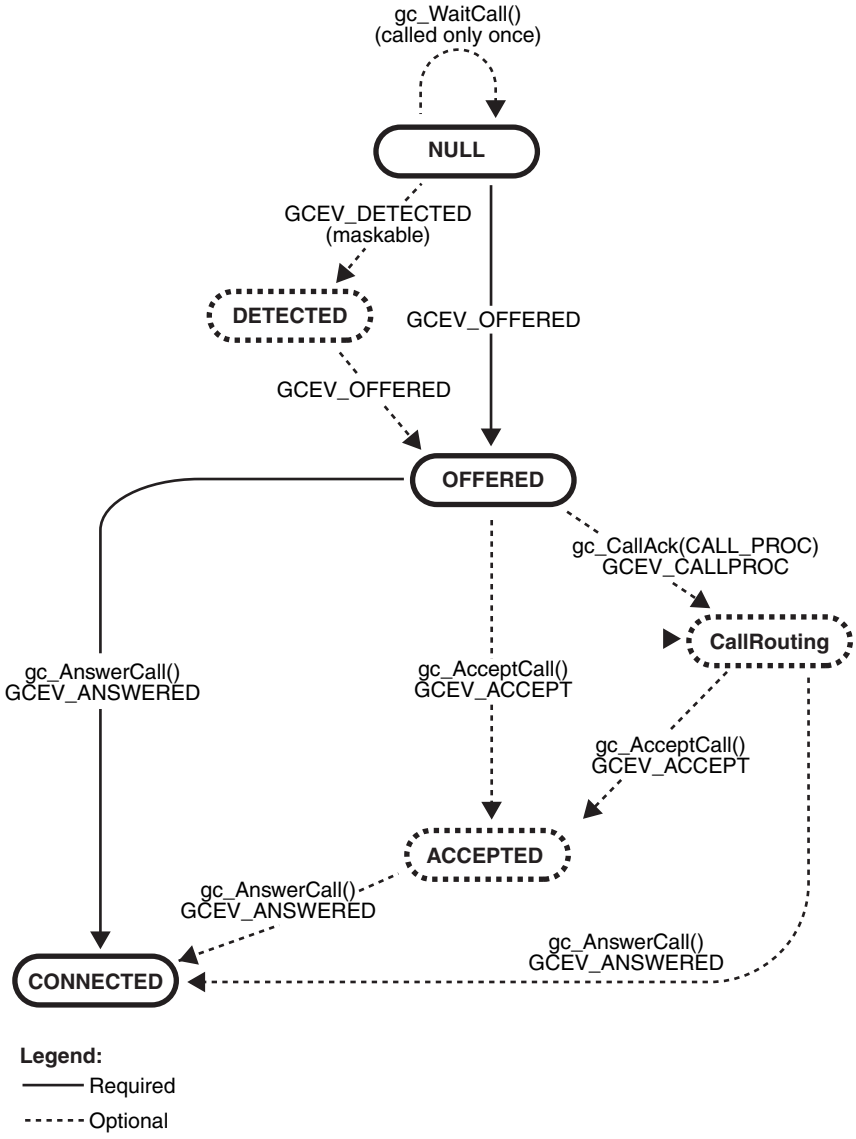


Table 6. Asynchronous Inbound Call State Transitions

State Description	Previous/Next State	Valid Call State Transition Functions	Call Transition Events
Accepted (GCST_ACCEPTED) Maskable	**Previous**: Offered, CallRouting **Next**: GCEV_ANSWERED -> Connected state GCEV_DISCONNECTED -> Disconnected state GCEV_DROP CALL -> Idle state	**gc_AnswerCall( )**, **gc_DropCall( )**	GCEV_DISCONNECTED, GCEV_DROPCALL or GCEV_ANSWERED
Call Routing (GCST_CALLROUTING) Maskable	**Previous**: Offered **Next**: GCEV_ANSWERED -> Connected state GCEV_ACCEPT -> Accepted state GCEV_DISCONNECTED -> Disconnected state GCEV_DROPCALL -> Idle state	**gc_AnswerCall( )**, **gc_AcceptCall( )**, **gc_DropCall( )**	GCEV_DISCONNECTED, GCEV_DROPCALL, GCEV_ACCEPT or GCEV_ANSWERED
Connected (GCST_CONNECTED) Not Maskable	**Previous**: Accept, Offered, CallRouting, Dialing, Proceeding, Alerting **Next**: GCEV_DISCONNECTED -> Disconnected state GCEV_DROPCALL -> Idle state	**gc_DropCall( )**	GCEV_DISCONNECTED GCEV_DROPCALL
Detected (GCST_DETECTED) Maskable	**Previous**: Null **Next**: GCEV_OFFERED -> Offered state GCEV_DISCONNECTED -> Disconnected state GCEV_DROPCALL -> Idle state	**gc_DropCall( )**	GCEV_DISCONNECTED, GCEV_DROPCALL, GCEV_OFFERED,

Table 6. Asynchronous Inbound Call State Transitions

State Description	Previous/Next State	Valid Call State Transition Functions	Call Transition Events
Null (GCST_NULL) Not Maskable	**Previous**: Idle **Next**: **gc_WaitCall( )** -> Null state **gc_ResetLineDev( )** -> Null state GCEV_OFFERED -> Offered state GCEV_DETECTED -> Detected state	**gc_WaitCall( )**	GCEV_DETECTED, GCEV_OFFERED
Offered (GCST_OFFERED) Not Maskable	**Previous**: Null, Detected **Next**: GCEV_ANSWERED -> Connected state GCEV_ACCEPT -> Accepted state GCEV_CALLPROC -> CallRouting state GCEV_DISCONNECTED -> Disconnected state GCEV_DROPCALL -> Idle state	**gc_CallAck( )**, **gc_AnswerCall( )**, **gc_AcceptCall( )**, **gc_DropCall( )**	GCEV_DISCONNECTED, GCEV_DROPCALL, GCEV_ACCEPT, GCEV_ANSWERED or GCEV_CALLPROC

The following sections describe the asynchronous inbound call processes.

3.4.1.2 Channel Initialization

To establish calls, the following conditions must be met:

- The condition of the line device must be unblocked. When a channel is initially opened, the initial condition of a line device is blocked. A “blocking” condition on a line device is indicated by the reception of a GCEV_BLOCKED event and an “unblocking” condition on a line device is indicated by the reception of a GCEV_UNBLOCKED event. The GCEV_BLOCKED and GCEV_UNBLOCKED events are sent as unsolicited events to the application in response to blocking alarms. GCEV_BLOCKED and GCEV_UNBLOCKED events are related to layer 1 alarms. GCEV_BLOCKED and GCEV_UNBLOCKED are used as what might be termed *flow-control events* within the application. For more information on blocking alarms and the GCEV_BLOCKED and GCEV_UNBLOCKED events, see Section 4.3, “Blocked and Unblocked Event Handling”. When the condition of the line device is unblocked, the line device is ready for establishing calls.
- The call state of the channel must be in the Null state. This is the initial call state of a line device when it is first opened. This state is also reached when a call is released or after the channel is reset.

If the above conditions are met, the application or thread must issue a **gc_WaitCall( )** function in the Null state to indicate readiness to accept an inbound call request on the specified line device. In the asynchronous mode, the **gc_WaitCall( )** function must be called only once after the line device

is opened using the **gc_OpenEx( )** function. However, if the **gc_ResetLineDev( )** function was issued, **gc_WaitCall( )** must be reissued. In asynchronous mode, it is not necessary to issue **gc_WaitCall( )** again after a call is released.

Note: After **gc_WaitCall( )** is issued to wait for incoming calls on a line device, it is possible to use **gc_makeCall( )** to make an outbound calls on that line device.

3.4.1.3 Call Detection

The inbound call from the network is received on the line device specified in the **gc_WaitCall( )** function, but the call has not been offered to the application. The technology call control layer typically sends an acknowledgement to the remote side. In some configurations, this acknowledgement can also be sent by the application when the call is offered. At this stage, the call is being processed, which typically involves allocating resources or waiting for more information. The GCEV_DETECTED event is generated, if enabled. If the GCEV_DETECTED event is generated, a new CRN is assigned to the incoming call. This event is for informational purposes to reduce glare conditions as the application is now aware of the presence of a call on the channel.

Note: When developing applications, if the GCEV_DETECTED event is not supported, a GCEV_DISCONNECTED event is only received if the host application already received the GCEV_OFFERED event before the remote side disconnects.

3.4.1.4 Call Offered

When an incoming call is received in en-bloc mode, where all the information required is available, the call is offered to the application by generating an unsolicited GCEV_OFFERED event (equivalent to a “ring detected” notification). This GCEV_OFFERED event causes the call to change to the Offered state. In the Offered state, a CRN is assigned as a means of identifying the call on a specific line device. If a GCEV_DETECTED event was generated before the GCEV_OFFERED event, the same CRN is assigned as the one assigned when the GCEV_DETECTED event was generated.

A call proceeding indication can be sent by the technology call control layer, or by the application by issuing the **gc_CallAck(GCACK_SERVICE_PROC)** function. Otherwise the application can accept or answer the call by issuing the **gc_AcceptCall( )** or **gc_AnswerCall( )** functions, respectively.

Note: When developing applications, if the GCEV_DETECTED event is not supported, a GCEV_DISCONNECTED event is only received if the host application already received the GCEV_OFFERED event before the remote side disconnects.

3.4.1.5 Call Routing

After the call has been offered, a call proceeding indication can be sent to the remote party to indicate that all the information has been received and the call is now proceeding. This indication can be sent by the technology call control layer or by the application by issuing the **gc_CallAck(GCACK_SERVICE_PROC)** function. This stage typically involves routing the call to the destination exchange or party. An information call routing tone can be played at this point to inform the remote party that the call is routing.

3.4.1.6 Call Acceptance

If the application or thread is not ready to answer the call, a **gc_AcceptCall( )** function is issued to indicate to the remote end that the call was received but not yet answered. This provides an interval during which the system can verify parameters, determine routing, and perform other tasks before connecting the call. A GCEV_ACCEPT event is generated when the **gc_AcceptCall( )** function is successfully completed and the call changes to the Accepted state. The application can then answer the call by issuing the **gc_AnswerCall( )** function.

3.4.1.7 Call Establishment

When the call is to be directly connected, such as to a voice messaging system, or if the application or thread is ready to answer the call, a **gc_AnswerCall( )** function is issued to make the final connection. Upon answering the call, a GCEV_ANSWERED event is generated and the call changes to the Connected state. At this point, the call is connected to the called party and call charges begin.

3.4.1.8 Call Failure

The following are various causes of call failures:

Call Rejection
: From the Offered state, the application or thread may reject the call by issuing the **gc_DropCall( )** function followed by a **gc_ReleaseCallEx( )** function (see the *Global Call API Library Reference*).

Task Failure
: If a call fails at any point in the call establishment process, that is, if a GCEV_TASKFAIL event is received by the application, the call stays in its current state. In most cases, the application needs to drop and release the call to return the line device to the Null state. However, in some cases, such as call failure due to a trunk error, the application needs to use the **gc_ResetLineDev( )** function to reset the line device to the Null state. For more information, see the **gc_DropCall( )**, **gc_ReleaseCallEx( )** and **gc_ResetLineDev( )** function descriptions in the *Global Call API Library Reference*.

3.4.1.9 Abandoned Calls

During call establishment, the remote side may choose to hang up before call setup has been completed. The application must be capable of handling error conditions and the lack of complete information when requesting call information.

Note: The GCEV_DETECTED event is not supported. If the host application has not received a GCEV_OFFERED event when the call is disconnected by the remote side, the host application will not receive any event. If the host application has already received a GCEV_OFFERED event, it receives a GCEV_DISCONNECTED event when the call is disconnected.

3.4.1.10 Inbound Call Scenarios in Asynchronous Mode

This section shows various asynchronous inbound call scenarios. For call scenarios used for a specific signaling protocol, check the *Global Call IP Technology Guide*.

Figure 4 shows a basic asynchronous call scenario for an incoming call.

Figure 4. Basic Asynchronous Inbound Call Scenario

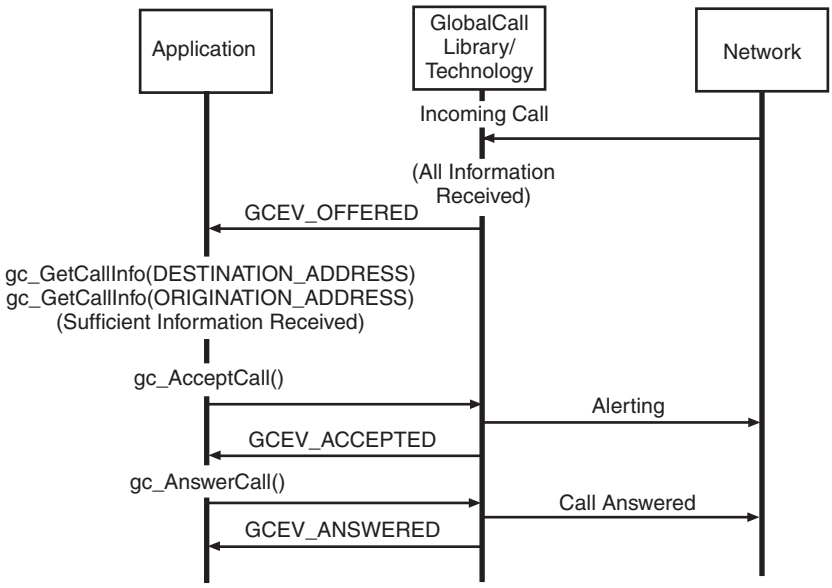


Figure 5 shows an asynchronous call scenario for an incoming call with call proceeding.

Figure 5. Incoming Call Scenario with Call Proceeding

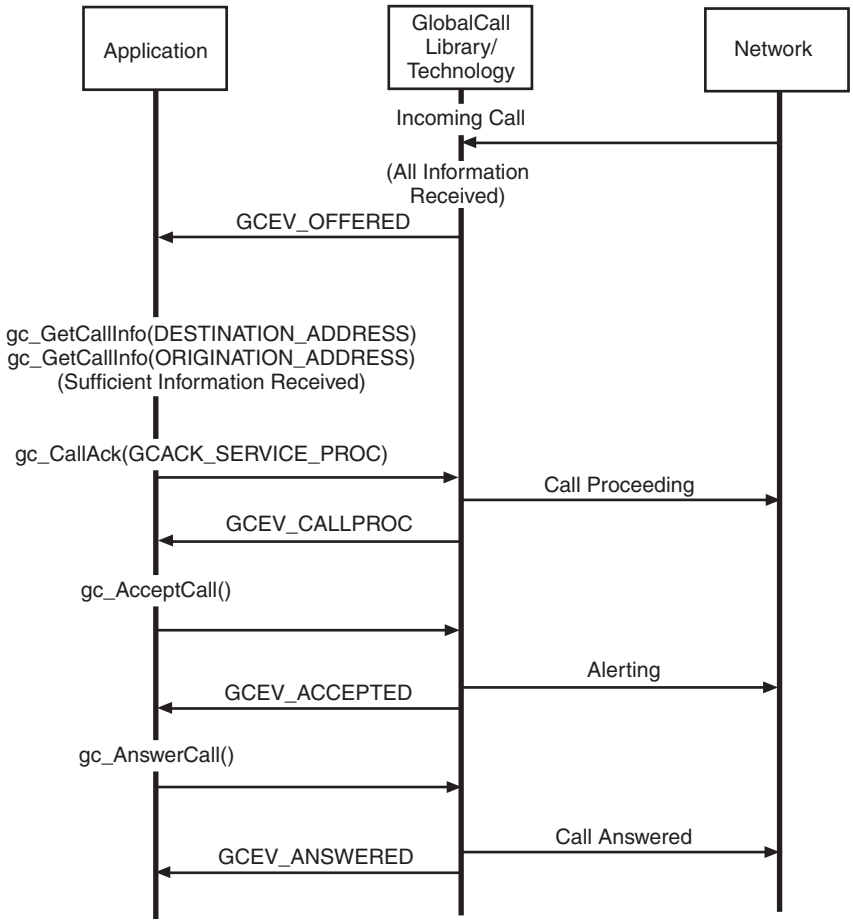


Figure 6 shows an asynchronous call scenario for an incoming call with call acknowledgement and call proceeding controlled by the call control layer.

Figure 6. Call Acknowledgement and Call Proceeding Done at Technology Call Control Layer

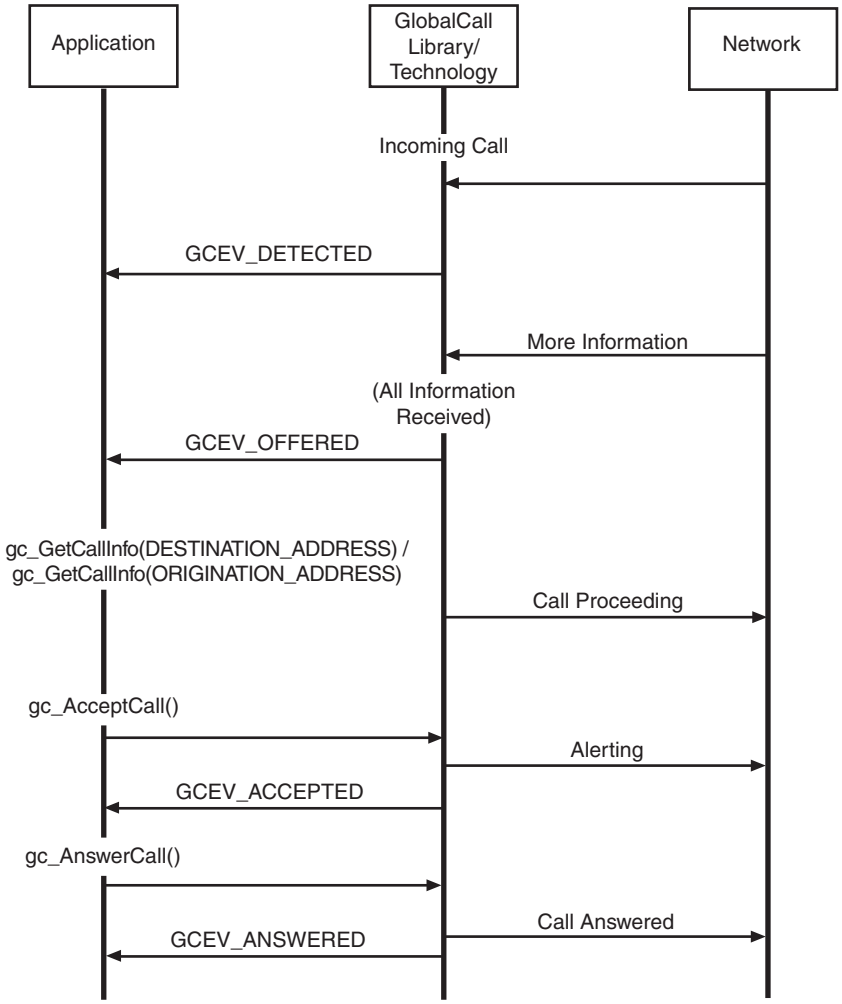
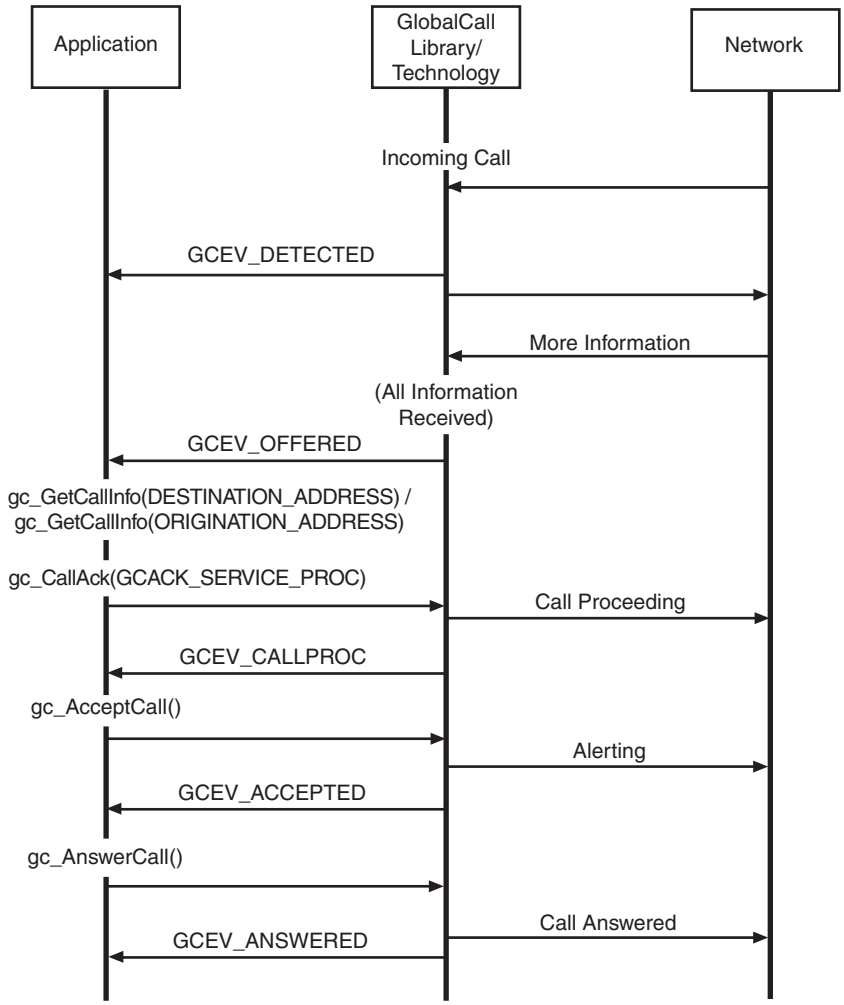


Figure 7 shows an asynchronous call scenario for an incoming call with call acknowledgement controlled by the call control layer and call proceeding controlled by the application.

Figure 7. Call Acknowledgement Done by the Technology Call Control Layer and Call Proceeding Done by the Application



3.4.2 Outbound Calls in Asynchronous Mode

This section describes how calls are established and shows call scenarios for asynchronous outbound calls. The following topics describe the processing of outbound calls in asynchronous mode:

- Outbound Calls in Asynchronous Mode Overview
- Channel Initialization
- Call Dialing
- Call Proceeding
- Call Alerting
- Call Connected
- Call Failure
- Outbound Call Scenarios in Asynchronous Mode

3.4.2.1 Outbound Calls in Asynchronous Mode Overview

Figure 8 illustrates a basic Outbound Call Model, which shows the call states associated with establishing a call in the asynchronous mode. All calls start from a Null state. The call establishment process for outbound calls is shown. Table 7 presents a summary of the outbound call state transitions.

Figure 8. Basic Asynchronous Outbound Call State Diagram

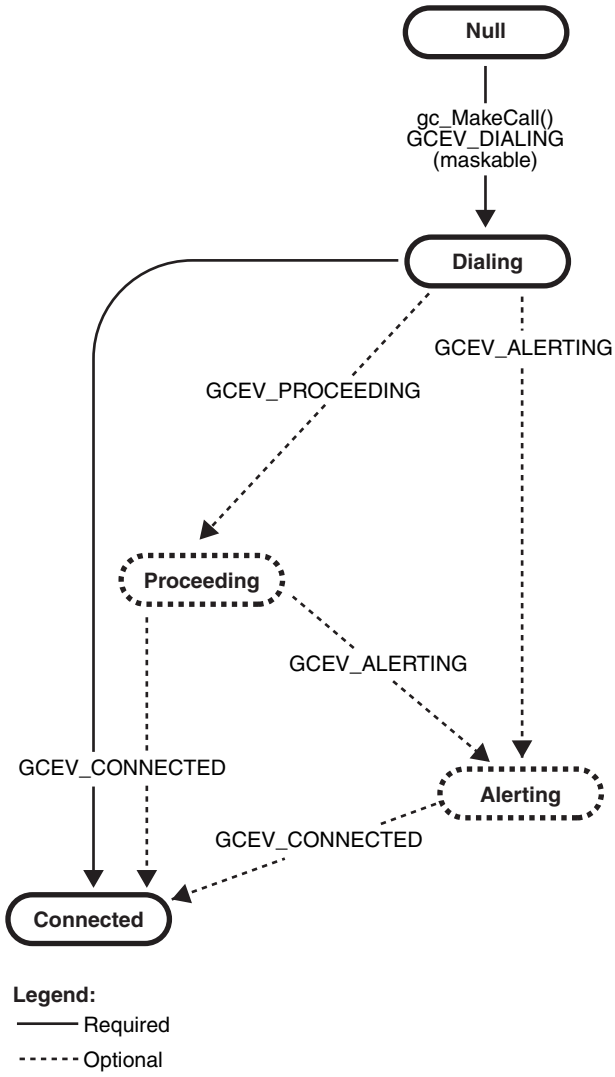


Table 7. Asynchronous Outbound Call State Transitions

State	Previous/Next State	Valid Call State Transition Functions	Call Transition Events
Alerting (GCST_ALERTING) Maskable	**Previous**: Proceeding, Dialing **Next**: GCEV_CONNECTED -> Connected state GCEV_DISCONNECTED -> Disconnected state GCEV_DROPCALL -> Idle state	**gc_DropCall( )**	GCEV_DISCONNECTED, GCEV_DROPCALL GCEV_CONNECTED
Dialing (GCST_DIALING) Not Maskable	**Previous**: Null **Next**: GCEV_CONNECTED -> Connected state GCEV_ALERTING -> Alerting (Delivered) state GCEV_PROCEEDING -> Proceeding state GCEV_DISCONNECTED -> Disconnected state GCEV_DROPCALL -> Idle state	**gc_DropCall( )**	GCEV_CONNECTED, GCEV_ALERTING, GCEV_PROCEEDING,GCEV_DISCONNECTED, GCEV_DROPCALL
Null (GCST_NULL) Not Maskable	**Previous**: Idle **Next**: **gc_ResetLineDev( )** -> Null GCEV_DIALING -> Dialing state GCEV_DETECTED -> Detected state	**gc_MakeCall( )**	GCEV_DIALING
Proceeding (GCST_PROCEEDING) Maskable	**Previous**: Dialing **Next**: GCEV_ALERTING -> Alerting (Delivered) state GCEV_CONNECTED -> Connected state GCEV_DISCONNECTED -> Disconnected state GCEV_DROPCALL ->Idle state	**gc_DropCall( )**	GCEV_DISCONNECTED, GCEV_DROPCALL, GCEV_CONNECTED, GCEV_ALERTING

The following sections describe the asynchronous outbound call processes, as shown in Figure 8, “Basic Asynchronous Outbound Call State Diagram”, on page 47.

3.4.2.2 Channel Initialization

To establish calls, the following conditions must be met:

- The condition of the line device must be unblocked. When a channel is initially opened, the initial condition of a line device is blocked. A “blocking” condition on a line device is indicated by the reception of a GCEV_BLOCKED event and an “unblocking” condition on a line device is indicated by the reception of a GCEV_UNBLOCKED event. The GCEV_BLOCKED and GCEV_UNBLOCKED events are sent as unsolicited events to the application in response to blocking alarms. (For more information on blocking alarms and the GCEV_BLOCKED and GCEV_UNBLOCKED events, see Section 4.3, “Blocked and Unblocked Event Handling”). When the condition of the line device is unblocked, the line device is ready for establishing calls.
- The call state of the channel must be in the Null state. This is the initial call state of a line device when it is first opened. This state is also reached when a call is released or after the channel is reset by issuing the **gc_ResetLineDev( )** function.

If the above conditions are met, the application is ready to make outbound calls.

3.4.2.3 Call Dialing

To initiate an outbound call using the asynchronous mode, the application issues a **gc_MakeCall( )** function that requests an outgoing call to be made on a specific line device. The **gc_MakeCall( )** function returns immediately. and the call state transitions to the Dialing state. The GCEV_DIALING event is generated (if enabled) to indicate that the call has transitioned to the Dialing state. A CRN is assigned to the call being established on that line device. If the **gc_MakeCall( )** function fails, the line device remains in the Null state. In this state, dialing information is sent to the remote side.

3.4.2.4 Call Proceeding

In the Dialing state, the remote side may indicate that all the information was received and the call is proceeding. In this case, the GCEV_PROCEEDING event is generated and the call transitions to the Proceeding state. The remote side may either accept or answer the call.

3.4.2.5 Call Alerting

If the remote end is not ready to answer the call, a GCEV_ALERTING event is generated. This event indicates that the called party has accepted but not answered the call and that the network is waiting for the called party to complete the connection. At this stage, the remote side is typically ringing. This GCEV_ALERTING event changes the call state to the Alerting state.

3.4.2.6 Call Connected

When the called party immediately accepts the call, such as a call directed to a FAX or voice messaging system, a GCEV_CONNECTED event is generated to indicate that the connection was established. This event changes the call to the Connected state. In the Connected state, the call is connected to the called party and call charges begin.

When the call is answered (the remote end makes the connection), a GCEV_CONNECTED event changes the call to the Connected state. In the Connected state, the call is connected to the called party and call charges begin. The GCEV_CONNECTED event indicates successful completion of the **gc_MakeCall( )** function.

3.4.2.7 Call Failure

The following are two causes of call failures:

Call Rejection

When the remote end does not answer the call, a GCEV_DISCONNECTED event is generated. This event is also generated when an inbound call arrives while the application is setting up an outbound call, causing a "glare" condition. Unless the protocol specifies otherwise, the incoming call takes precedence over the outbound call. When an asynchronous **gc_MakeCall( )** function conflicts with the arrival of an inbound call, all the resources need to be released for the outbound call. Subsequently, the GCEV_DISCONNECTED event is generated with a result value indicating that an inbound call took precedence. The **gc_DropCall( )** function must be issued after the GCEV_DISCONNECTED event is received.

If a **gc_MakeCall( )** function is issued while the inbound call is being set up, the **gc_MakeCall( )** function fails. The inbound call event is held in the driver until the CRN of the outbound call is released using the **gc_ReleaseCallEx( )** function. After release of the outbound CRN, the pending inbound call event is sent to the application. This behavior may be modified by the individual protocol specification.

Task Failure

If the **gc_MakeCall( )** cannot be completed successfully, a GCEV_TASKFAIL event or a GCEV_DISCONNECTED event is sent to the application. The result value associated with the event indicates the reason for the event. If the GCEV_TASKFAIL event is sent, then a problem occurred when placing the call from the local end.

3.4.2.8 Outbound Call Scenarios in Asynchronous Mode

This section shows various asynchronous outbound call scenarios. For call scenarios used for a specific signaling protocol, check the *Global Call IP Technology Guide*.

Figure 9 shows a basic asynchronous call scenario for outgoing calls.

Figure 9. Asynchronous Outbound Call Scenario

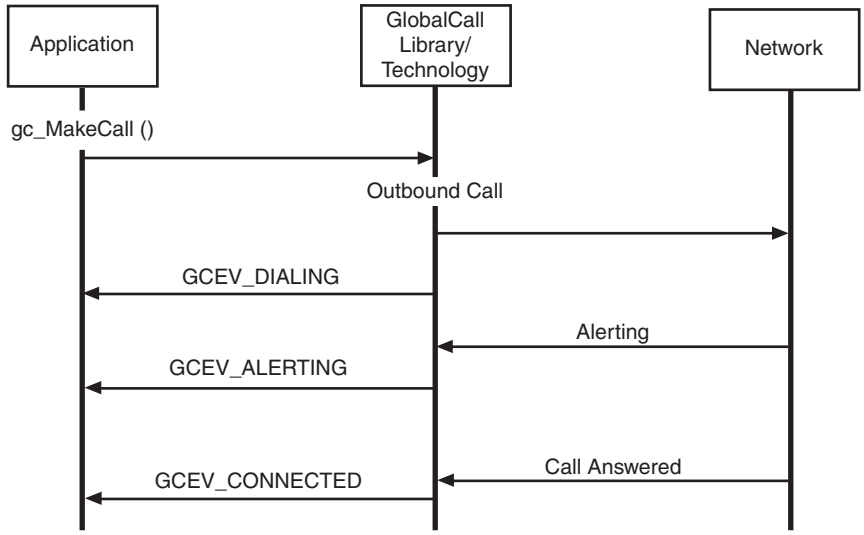
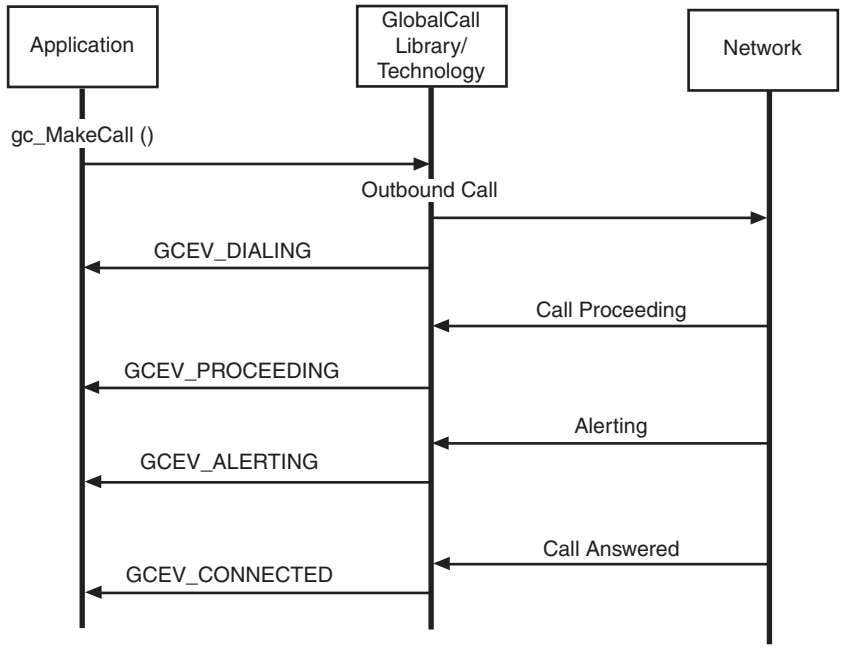


Figure 10 shows an asynchronous call scenario for outgoing calls with call acknowledgement.

Figure 10. Asynchronous Outbound Call Scenario With Call Acknowledgement



3.4.3 Call Termination in Asynchronous Mode

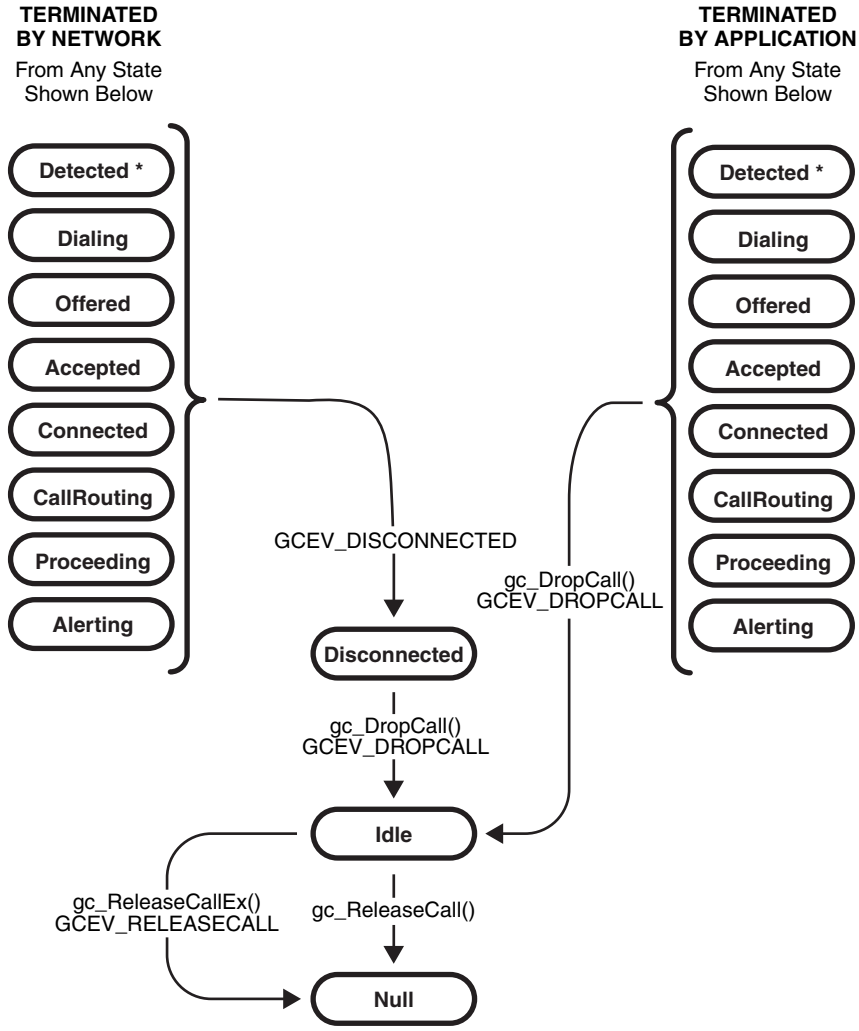
This section describes how calls are terminated and shows call scenarios for asynchronous call termination. The following topics describe call termination in asynchronous mode:

- Call Termination in Asynchronous Mode Overview
- User Initiated Termination
- Network Initiated Termination
- Call Release
- Call Termination Call Control Scenarios in Asynchronous Mode

3.4.3.1 Call Termination in Asynchronous Mode Overview

Figure 11 illustrates the call states associated with call termination or call teardown in the asynchronous mode initiated by either a call disconnection or failure. See Table 8 for a summary of the call state transitions. A call can be terminated by the application or by the detection of a call disconnect from the network. Either of these terminations can occur at any point in the process of setting up a call and during any call state.

Figure 11. Asynchronous Call Tear-Down State Diagram



Note: * applies if the application requested to be notified of GCEV_DETECTED events.

Table 8. Asynchronous Call Termination Call State Transitions

State	Previous/Next State	Valid Call State Transition Functions	Call Transition Events
Disconnected (GCEV_DISCONNECTED) Not maskable	**Previous**: Offered, Accepted, Connected, Dialing, Proceeding, Alerting, CallRouting **Next**: GCEV_DROPCALL -> Idle state	**gc_DropCall( )**	GCEV_DROPCALL
Idle (GCST_IDLE) Not Maskable	**Previous**: Offered, Accepted, Connected, Dialing, Proceeding, Alerting, CallRouting, Disconnected **Next**: GCEV_RELEASECALL -> Null	**gc_ReleaseCallEx( )**	GCEV_RELEASECALL

3.4.3.2 User Initiated Termination

The application terminates a call by issuing a **gc_DropCall( )** function that initiates disconnection of the call specified by the CRN. When the remote side responds by disconnecting the call, a GCEV_DROPCALL event is generated and causes a transition from the current call state to the Idle state. The user must then issue the **gc_ReleaseCallEx( )** function to release all internal resources allocated for the call.

3.4.3.3 Network Initiated Termination

When a network call termination is initiated, an unsolicited GCEV_DISCONNECTED event is generated. This event indicates the call was disconnected at the remote end or an error was detected, which prevented further call processing. The GCEV_DISCONNECTED event causes the call state to change from the current call state to the Disconnected state. This event may be received during call setup or after a connection is requested. In the Disconnected state, the user issues the **gc_DropCall( )** function to disconnect the call. The **gc_DropCall( )** function is equivalent to *set hook ON*. After the remote side is notified about the call being dropped, a GCEV_DROPCALL event is generated causing the call state to change to the Idle state. In the Idle state, the **gc_ReleaseCallEx( )** function must be issued to release all internal resources committed to servicing the call.

3.4.3.4 Call Release

Once in the Idle state, the call has been disconnected and the application must issue a **gc_ReleaseCallEx( )** function to free the line device for another call. The **gc_ReleaseCallEx( )** function releases all internal system resources committed to servicing the call. A GCEV_RELEASECALL event is generated and the call state transitions to the Null state.

3.4.3.5 Call Termination Call Control Scenarios in Asynchronous Mode

This section shows various asynchronous call termination call scenarios. For call scenarios used for a specific signaling protocol, check the *Global Call IP Technology Guide*.

Figure 12 shows an asynchronous user initiated call termination scenario.

Figure 12. User Initiated Asynchronous Call Termination Scenario

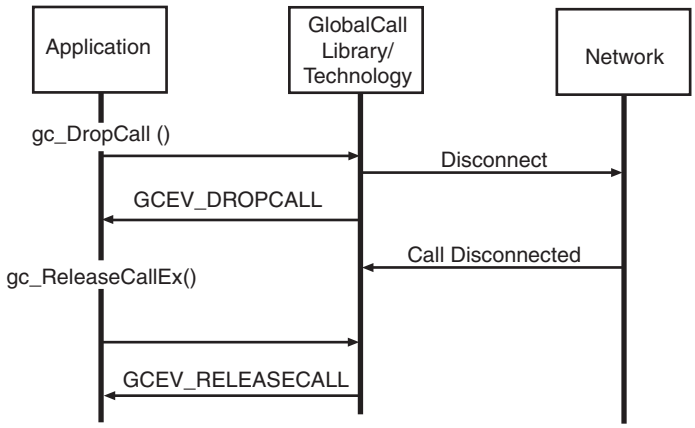
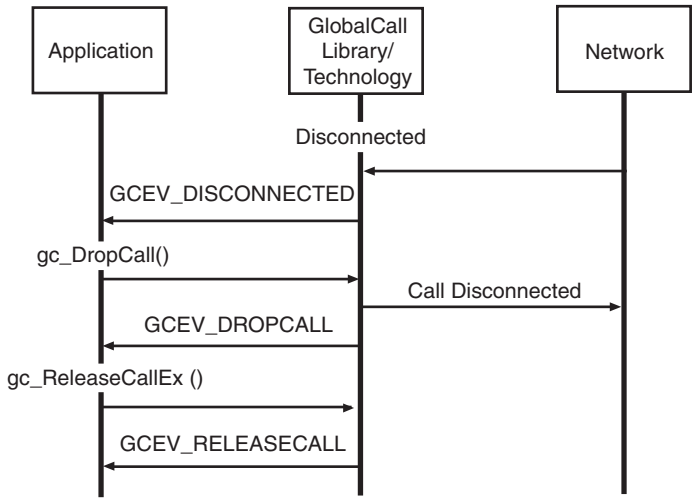


Figure 13 shows an asynchronous network initiated call termination scenario.

Figure 13. Network Initiated Asynchronous Call Termination Scenario



3.4.4 Handling Unsolicited Events

The application must handle unsolicited events in the synchronous mode, unless these events are masked or disabled. The **gc_SetConfigData( )** function specifies the events that are enabled or disabled for a specified line device. This function sets the event mask associated with the specified line device. If an event bit in the mask is cleared, the event is disabled and not sent to the application.

The unsolicited events listed in Table 9 require a signal handler if they are enabled. Unsolicited events that cannot be masked must use a signal handler. All IP-specific unsolicited events also require a signal handler (see the *Global Call IP Technology Guide* for details). If any of these unsolicited events are not masked by the application and signal handlers are not defined, they are queued without being retrievable and memory problems are likely to occur.

Table 9. Unsolicited Events Requiring Signal Handlers

Event	Default Setting	Maskable
GCEV_ALERTING	enabled	yes
GCEV_PROCEEDING	disabled	yes
GCEV_DETECTED	disabled	yes
GCEV_BLOCKED	enabled	yes
GCEV_UNBLOCKED	enabled	yes
GCEV_DISCONNECTED	enabled	no
GCEV_TASKFAIL	enabled	no

Event Handling 4

This chapter describes how Global Call handles events generated in the call state model. Topics include:

4.1 Overview of Event Handling

The Global Call protocol handler continuously monitors the line device for events from the network. As each call is processed through its various states, corresponding events are generated and passed to the application. An overview of Global Call event categories is provided in this chapter. Specific event definitions are described in the *Global Call API Library Reference*. See the *Global Call IP Technology Guide* for IP-specific event information.

4.2 Event Categories

The events that can occur when using the Global Call API are divided into the following categories:

Termination
: Events returned after the termination of a function. Termination events apply to asynchronous programming only.

Notification
: Events that are requested by the application and provide information about a function call. Notification events apply to synchronous and asynchronous programming.

Unsolicited
: Events triggered by, and providing more information about, external events. Unsolicited events apply to synchronous and asynchronous programming.

See the *Global Call API Library Reference* for detailed information about each event and the *Global Call IP Technology Guide* for IP-specific event information.

4.3 Blocked and Unblocked Event Handling

Global Call uses the concept of *blocked* and *unblocked* conditions for line devices. By default, when the **gc_OpenEx( )** function is used to open a line device, the line device is in a blocked condition meaning that the application can not perform call related functions on the line device, such as waiting for a call or making a call. The application must wait for the GCEV_UNBLOCKED event before waiting for a call or making a call.

Note: Since, by default, the line device is initially in the blocked condition, the application does **not** receive an initial GCEV_BLOCKED event.

Circumstances can occur, such as a blocking layer 1 (physical) alarm or the remote side going out of service, that cause a line device to move to a blocked condition. When this happens, the application receives a GCEV_BLOCKED event. When the line device is in the blocked condition, the application can only perform a small subset of the valid functions for line devices. The functions common to all interface technologies and that can be used while a line device is in the blocked condition are:

- **gc_DropCall( )**
- **gc_ReleaseCallEx( )**
- **gc_Close( )**
- Functions related to alarm processing and retrieving alarm information, for example, **gc_AlarmName( )**
- Functions related to error processing, for example, **gc_ErrorInfo( )**
- Functions related to event processing, for example, **gc_ResultInfo( )**, **gc_GetMetaevent( )**and **gc_GetMetaeventEx( )**
- Functions related to retrieving information about the call control libraries, for example, **gc_CCLibIDToName( )**
- **gc_AttachResource( )** and **gc_Detach( )**

As indicated in the list above, the application may drop and release calls while a line device is in the blocked condition, but it should **not** do so in response to the GCEV_BLOCKED event. If a call is active, typically a GCEV_DISCONNECTED event arrives either just before or just after the GCEV_BLOCKED event, at which point the application should drop and release the call indicated by the GCEV_DISCONNECTED event.

Note: The Global Call term *blocked* does not refer to the signaling bits indicating a blocked condition as defined in some network interface technologies, although the line device may move to a blocked condition as a consequence of the signaling bits indicating a blocked condition.

At some point, the application may receives a GCEV_UNBLOCKED event, indicating that the conditions blocking a line device have been removed and the line device has now returned to the unblocked condition. The application can once again use any valid function on the line device.

The reception of the GCEV_BLOCKED and GCEV_UNBLOCKED events may be disabled using the **gc_SetConfigData( )** function. The default is that these events are enabled. However, disabling the reception of these events is **not** recommended since the application will not be notified of these critical events. In addition, if the GCEV_BLOCKED event is disabled, some functions will fail with a reason of EGC_INVALIDSTATE, which may cause confusion. For more information on

blocking alarms and the GCEV_BLOCKED and GCEV_UNBLOCKED events, see Section 8.2.1, "Generation of Events for Blocking Alarms", on page 75.

Note: A GCEV_UNBLOCKED event will be generated when opening a virtual board device. A GCEV_BLOCKED event will also be generated if there are blocking alarms on the virtual board and the corresponding GCEV_UNBLOCKED event will be generated when the blocking alarms clear. The application must be prepared to handle these events.

4.4 Event Retrieval

All events are retrieved using the current Standard Runtime Library (SRL) event retrieval mechanisms (see the *Standard Runtime Library API Programming Guide* for details), including event handlers. The **gc_GetMetaEvent( )** function or, for Windows extended asynchronous models, the **gc_GetMetaEventEx( )** function, maps the current SRL event into a metaevent. A metaevent is a data structure that explicitly contains the information describing the event. This data structure provides uniform information retrieval among all call control libraries.

For Global Call events, the structure contains Global Call related information (CRN and line device) used by the application. For events that are not Global Call events, the device descriptor, the event type, a pointer to variable length event data, and the length of the event data are available through the METAEVENT structure. Since all the data associated with an event is accessible via the METAEVENT structure, no additional SRL calls are required to access the event data.

The LDID associated with an event is available from the linedev field of the METAEVENT. If the event is related to a CRN, that CRN is available from the crn field of the METAEVENT; if the crn field of the METAEVENT is 0, then the event is not a call-related event.

The METAEVENT structure also includes an extevtdatap field which contains a pointer to more information about the event. The memory pointed to by the extevtdatap field should be treated as **read-only** and should not be altered and/or freed.

The application should issue a **gc_DropCall( )** function before issuing the **gc_ReleaseCallEx( )** function. Failure to issue this function could result in one or more of the following problems:

- memory problems due to memory being allocated and not being released
- a blocking condition
- events sent to the previous user of a CRN that could be processed by a later user of the CRN with unexpected results

The reason for an event can be retrieved using the **gc_ResultInfo( )** function. The information returned uniquely identifies the cause of the event.

4.5 Events Indicating Errors

Events that explicitly provide error indications are as follows:

GCEV_TASKFAIL
: Received when an API function call fails

When this event is received, the application should call **gc_ResultInfo( )** immediately after the event arrives to determine the reason for the event. The data structure associated with **gc_ResultInfo( )** can contain reason information provided by Global Call and additional reason information provided by the underlying call control library. See the *Global Call API Library Reference* for more information.

4.6 Masking Events

Some events are maskable. See the **gc_SetConfigData( )** function description in the *Global Call API Library Reference* for specific information regarding enabling and disabling events.

4.7 Event Handlers

An event handler is a user-defined function called by the SRL to handle a specific event that occurs on a specified device.

Typically, in a Windows environment, processing events within a thread or using a separate thread to process events tends to be more efficient than using event handlers. However, if event handlers are used, such as when an application is being ported from Linux, then you must use the asynchronous with SRL callback model.

The following guidelines apply to using event handlers:

- more than one handler can be enabled for an event. The SRL calls **all** specified handlers when the event is detected
- handlers can be enabled or disabled from any thread
- general handlers can be enabled to handle **all** events on a specific device
- a handler can be enabled to handle **any** event on **any** device
- synchronous functions cannot be called from a handler

By default, when the **sr_enbhdlr( )** function is first called, a thread internal to the SRL is created to service the application-enabled event handlers. This SRL handler thread exists as long as one handler is still enabled. The creation of this internal SRL event handler thread is controlled by the SR_MODELTYPE value of the SRL **sr_setparm( )** function. The SRL handler thread should be:

- enabled when using the asynchronous with SRL callback model. Enable the SRL event handler thread by **not** specifying the SR_MODELTYPE value (default is to enable) or by setting this value to SR_MTASYNC (do **not** specify SR_STASYNC).
- disabled when using an application-handler thread wherein a separate event handler thread is created within the application that calls the **sr_waitevt( )** and **gc_GetMetaEvent( )** functions. For an application-handler model, use the asynchronous with SRL callback model **but** set the SR_MODELTYPE value to SR_STASYNC to disable the creation of the internal SRL event handler thread.

  Note: An application-handler thread must **not** call any synchronous functions.

See the *Standard Runtime Library API Programming Guide* for the hierarchy (priority) order in which event handlers are called.

Application Development Guidelines 5

This chapter provides some tips when developing programs using Global Call. Topics include:

5.1 General Programming Tips

The following tips apply when programming with Global Call:

- When using Global Call functions, the application must use the Global Call handles (that is, the line device ID and CRN) to access Global Call functions. Do not substitute a network, voice or media device handle for the Global Call line device ID or CRN. If the application needs to use a network, voice or media device handle for a specific network or voice library call, for example **dx_play( )**, you must use the **gc_GetResourceH( )** to retrieve the network, voice or media device handle, associated with the specified Global Call line device. The **gc_GetResourceH( )** function is only needed if the voice or media resource is associated with a Global Call line device. If a voice resource is not part of the Global Call line device, the device handle returned from the **dx_open( )** call should be used.
- Do not access the underlying call control libraries directly. All access **must** be done using the Global Call library, that is, using Global Call (**gc_**) functions.
- Do not call any network library (**dt_**) function directly from your application that may affect the state of the line or the reporting of events, for example, **dt_settssig( )**, **dt_setevtmsk( )**, or others.
- The GCEV_BLOCKED and the GCEV_UNBLOCKED events are line related events, not call related events. These events do not cause the state of a call to change.
- Before exiting an application:
  - Drop and release **all** active calls, using the **gc_DropCall( )** and **gc_ReleaseCallEx( )** functions.
  - Close **all** open line devices, using the **gc_Close( )** function.
  - Stop the application, using the **gc_Stop( )** function
- Before issuing **gc_DropCall( )**, you must use the **dx_stopch( )** function to terminate any application-initiated voice functions, such as **dx_play( )** or **dx_record( )**.
- In Windows environments, although asynchronous models are more complex than the synchronous model, asynchronous programming is recommended for more complex applications that require coordinating multiple tasks. Asynchronous programming can handle multiple channels in a single thread. In contrast, synchronous programming requires separate threads. Asynchronous programming uses system resources more efficiently because it handles multiple channels in a single thread. Asynchronous models let you program complex applications easily, and achieve a high level of resource management in your application by

combining multiple voice channels in a single thread. This streamlined code reduces the system overhead required for inter process communication and simplifies the coordination of events from many devices.

- In Windows environments, when calling the **gc_GetMetaEventEx( )** function from multiple threads, ensure that your application uses unique thread-related METAEVENT data structures or ensure that the METAEVENT data structure is not written to simultaneously.

5.2 General Considerations when Using Global Call

Global Call supports the development of applications that use HMP. The following topics provide guidelines for using Global Call with HMP:

- Routing Overview
- Working with Flexible Routing Configurations
- Handling Multiple Call Objects Per Channel in a Glare Condition

5.2.1 Routing Overview

The HMP software supports flexible routing configurations. With flexible routing, the resource devices (voice/fax/media) and network interface devices are independent, which allows exporting and sharing of the resources.

5.2.2 Working with Flexible Routing Configurations

The following topics provide more information about using Global Call with flexible routing configuration:

- Using Device Handles (Flexible Routing)
- Multi-Threading and Multi-Processing (Flexible Routing)
- Initializing an Application (Flexible Routing)
- Device Initialization Hint (Flexible Routing)

5.2.2.1 Using Device Handles (Flexible Routing)

Application performance may be a consideration when opening and closing devices using Global Call. If an application must use Global Call to dynamically open and close devices as needed, it can impact the application's performance. One way to avoid this is to open all devices during application initialization and keep them open for the duration of the application, closing them only at the end.

5.2.2.2 Multi-Threading and Multi-Processing (Flexible Routing)

The R4 APIs support multi-threading and multi-processing with some restrictions on multi-processing as follows:

- One specific channel can only be opened in one process at a time. There can, however, be multiple processes accessing different sets of channels. In other words, ensure that each process is provided with a unique set of devices to manipulate.
- If a channel was opened in process A and then closed, process B is then allowed to open the same channel. However, since closing a channel is an asynchronous operation when using R4, there is a small gap between the time when the xx_close( ) function returns in process A and the time when process B is allowed to open the same channel. If process B opens the channel too early, things could go wrong. For this reason, this type of sequence should be avoided.

5.2.2.3 Initializing an Application (Flexible Routing)

A device must first be opened in order to obtain its handle, which can then be used to access the device functionality. Since applications use Global Call for call control (that is, for call setup and tear-down), all Intel network interface devices must be opened using the **gc_OpenEx( )** function.

Once the call has been established, voice and or data streaming should be done using the Voice API. Functions such as **dx_playiottdata( )**, **dx_reciottdata( )**, and **dx_dial( )** can be used. Of course, in order to do so, the voice device handle must be obtained.

5.2.2.4 Device Initialization Hint (Flexible Routing)

In some applications, when **xx_open( )** functions (Global Call, Voice, Fax) are issued asynchronously, it may cause slow device-initialization performance. Fortunately, you can avoid this particular problem quite simply by reorganizing the way the application opens and then configures devices. The recommendation is to do all **xx_open( )** functions for all channels before proceeding with the next function. For example, you would have one loop through the system devices to do all the **xx_open( )** functions first, and then start a second loop through the devices to configure them, instead of doing one single loop where an **xx_open( )** is immediately followed by other API functions on the same device. With this method, by the time all **xx_open( )** commands are completed, the first channel will be initialized, so you won't experience problems.

This change is not necessary for all applications, but if you experience poor initialization performance, you can gain back speed by using this hint.

5.2.3 Handling Multiple Call Objects Per Channel in a Glare Condition

Global Call supports the handling of multiple call objects per channel in a glare condition. An application running on bi-directional circuits is capable of handling two CRNs on a single line device, where one call can be in an Idle state, while the other call is in Active state. For example, a glare condition occurs when a call has been dropped but not released and an inbound call is detected as indicated in Table 10. In order to avoid a long delay in processing the inbound call, the

Global Call library does not wait for the outbound call to be released before notifying the application of the inbound call.

Table 10. Handling Glare

Application	Global Call Library
gc_MakeCall(CRN1) -->	
	<-- GCEV_DISCONNECTED(CRN1)
gc_DropCall(CRN1) -->	
	<-- GCEV_OFFERED(CRN2)
gc_AcceptCall(CRN2) -->	
	<-- GCEV_DROPCALL(CRN1)
gc_ReleaseCallEx(CRN1) -->	

Alternatively, the application can just respond to events using their associated CRN, simply performing a **gc_ReleaseCallEx( )** upon reception of any GCEV_DROPCALL event whether the CRN is the active one or not. Using this procedure, the application only needs to store one CRN per line device.

Error Handling 6

The chapter describes the error handling capabilities provided by Global Call. Topics include the following:

6.1 Error Handling Overview

When an error occurs during execution of a function, one of the following occurs:

- The function returns with a value < 0
- The unsolicited error event, GCEV_TASKFAIL, is sent to the application

Call control libraries supported by the Global Call API may have a larger set of error codes than those defined in the *gcerr.h* header file. The call control library error values are available using the **gc_ErrorInfo( )** function, which retrieves Global Call and call control library information. To retrieve the information, this function must be called immediately after the Global Call function failed. This function returns a result value associated directly with the Global Call and call control library.

The **gc_ResultInfo( )** function retrieves information about solicited and unsolicited events when a Global Call application gets an expected or unexpected event. To retrieve the information, the **gc_ResultInfo( )** function must be called immediately after a Global Call event arrives and before the next event returns Global Call and call control library information related to the last Global Call function call. To process an error, this function must be called immediately after an event is returned to the application. For example, if an alarm occurs while making an outbound call, a GCEV_DISCONNECTED event is sent to the application with a result value indicating an alarm on the line. The GCEV_BLOCKED event is also generated with a result value that also indicates an alarm on the line. See the *Global Call IP Technology Guide*for information on specific protocol errors.

If an error occurs during execution of an asynchronous function, a termination event, such as the GCEV_SETCONFIGDATA_FAIL event is sent to the application. No change of state is triggered by this event. If events on the line require a state change, this state change occurs as described in Section 3.4.3, “Call Termination in Asynchronous Mode”, on page 52. When an error occurs during a protocol operation, the error event is placed in the event queue with the error value that identifies the error. Upon receiving a GCEV_TASKFAIL event, the application can retrieve the reason for the failure using the **gc_ResultInfo( )** function.

An unsolicited GCEV_ERROR event can be received if an internal component fails. The **gc_ResultInfo( )** function can be used to determine the reason for the event. Valid reasons are any of the Global Call reasons (error code or result values) or a call control library-specific reason (see the *Global Call IP Technology Guide*).

Call Control 7

This chapter describes Global Call capabilities relating to call control. Topics include:

7.1 Call Progress Analysis

When using IP technology, typically packetized messages are used to convey call analysis information. See the *Global Call IP Technology Guide* for more information.

7.2 Resource Routing

The **gc_GetResourceH( )** function can be used to obtain the media and voice device handles, respectively, associated with the specified line device. The **gc_GetResourceH( )** function, with a **resourcetype** of GC_MEDIADEVICE returns the media device handle for the specified line device. The **gc_GetResourceH( )** function, with a **resourcetype** of GC_VOICEDEVICE, returns the voice device handle only if the specified line device has a voice, media, or tone resource associated with it, for example, if a voice channel was specified in the **gc_OpenEx( )** function **devicename** parameter, or if the voice channel was subsequently attached to the line device and has remained attached to that line device.

Refer to the appropriate *Global Call IP Technology Guide* for IP-specific information on routing resources when using the **gc_OpenEx( )** function to specify a voice or media resource, or when using the **gc_AttachResource( )** function to associate a voice or media resource with a Global Call line device.

7.3 Feature Transparency and Extension

Global Call Feature Transparency and Extension (FTE) provides a common interface to multiple network interface specific libraries for features that are abstracted across multiple call control libraries (see Figure 1, “Global Call Architecture for IP Technology”, on page 19). FTE is described in the following topics:

- Feature Transparency and Extension Overview
- Technology-Specific Feature Access
- Technology-Specific User Information

7.3.1 Feature Transparency and Extension Overview

FTE is comprised of a number of Global Call functions. These functions provide the flexibility to extend the generic Global Call API to access all technology or protocol-specific features unique to any given network interfaces that were formerly only accessible via their native technology call control libraries. Thus, all technology-specific features may be accessible from the application solely via the singular Global Call library interface, thereby alleviating the need to access these call control libraries directly via additional APIs.

The Global Call API functions provided for FTE are:

gc_Extension( )
provides a generic interface extensible for technology-specific features

gc_SetUserInfo( )
permits technology-specific user information to be defined for the specified line device or call

Note: The **gc_SetUserInfo( )** function is not supported for a board device.

7.3.2 Technology-Specific Feature Access

The **gc_Extension( )** function provides a single common interface to access various technology-specific features supported by underlying call control libraries.

This Global Call function utilizes an extension function identifier (**ext_id**) to specify the feature. The *Global Call IP Technology Guide* lists all the supported extension function identifiers (**ext_id** values) and the associated features that are accessible via the **gc_Extension( )** function.

By specifying the associated parameter identifiers (also described in the *Global Call IP Technology Guide*), and either the target line device or a specific call, those features unique to IP technology may be utilized entirely using the Global Call API. Without FTE support, a Global Call application requiring this feature support would also have to be written to the specific call control API in addition to the Global Call API.

The **gc_Extension( )** function may be supported in either asynchronous mode, synchronous mode or both depending on the call control library.

If the **gc_Extension( )** function is supported and called in synchronous mode, the relevant information parameters returned in the GC_PARM_BLK buffer must be processed or copied prior to the next Global Call function call. The reason for this is that the GC_PARM_BLK buffer will be deallocated within Global Call in a subsequent function call.

If the **gc_Extension( )** function is supported and called in asynchronous mode, relevant information may be returned via the call control library via GCEV_EXTENSIONCMPLT termination event and its referenced extension block structure, EXTENSIONEVTBLK. The EXTENSIONEVTBLK structure contains IP-specific information and is referenced via the extevtdatap pointer in the METAEVENT structure associated with the GCEV_EXTENSIONCMPLT event. See the *Global Call API Library Reference* for more information about these structures.

The **gc_Extension( )** function can also be used to transmit information to the remote endpoint. In this case, while the application at the local end point receives a GCEV_EXTENSIONCMPLT, the application at the remote end point will receive an unsolicited GCEV_EXTENSION notification event from the network with the transmitted information. The EXTENSIONEVTBLK structure contains the transmitted information and is referenced via the extevtdatap pointer in the METAEVENT structure associated with the GCEV_EXTENSION event.

The application at the local end point may also receive an unsolicited GCEV_EXTENSION event with information from the network.

It is important to note that the EXTENSIONEVTBLK structure referenced in the GCEV_EXTENSION event has a persistence only until the next call of **gc_GetMetaEvent( )**. In other words, any information contained or referenced in the associated EXTENSIONEVTBLK structure must be either processed or copied in the application, or risk having the memory space containing the actual information lost on the next **gc_GetMetaEvent( )** call.

7.3.3 Technology-Specific User Information

The **gc_SetUserInfo( )** function permits the application to configure user information for the specified line device that is transmitted to or received from the remote side. The actual content and format of the user information is protocol-specific. Refer to the *Global Call IP Technology Guide* for details on the format of the user information supported and the proper usage of the **gc_SetUserInfo( )** function.

Alarm Handling 8

This chapter describes the Global Call Alarm Management System (GCAMS). Topics include the following:

8.1 Alarm Handling Overview

Global Call alarms originate from alarm source objects (ASO). An alarm source object can be a network library, or a call control library, or it can reside within a call control library. Some alarm source objects are for internal Global Call use only and are not available to the application.

There are basically two sources of Global Call alarms:

- Layer 1 alarms (physical alarms)
- "Logical" alarms, such as remote side out of service, or layer 2 or layer 3 out of service

The portion of the Global Call call control library that manages alarms is called the Global Call Alarm Management System (GCAMS). GCAMS is initialized automatically when Global Call is started.

GCAMS provides Global Call applications with the ability to receive extensive alarm information. Some of the ways this information can be used include:

- Managing the network
- Troubleshooting hardware
- Monitoring line quality
- Working with the central office to solve line problems
- Generating status reports
- Modifying alarm source object properties and characteristics based on alarm history
- Manual handling of alarms for drop and insert applications.

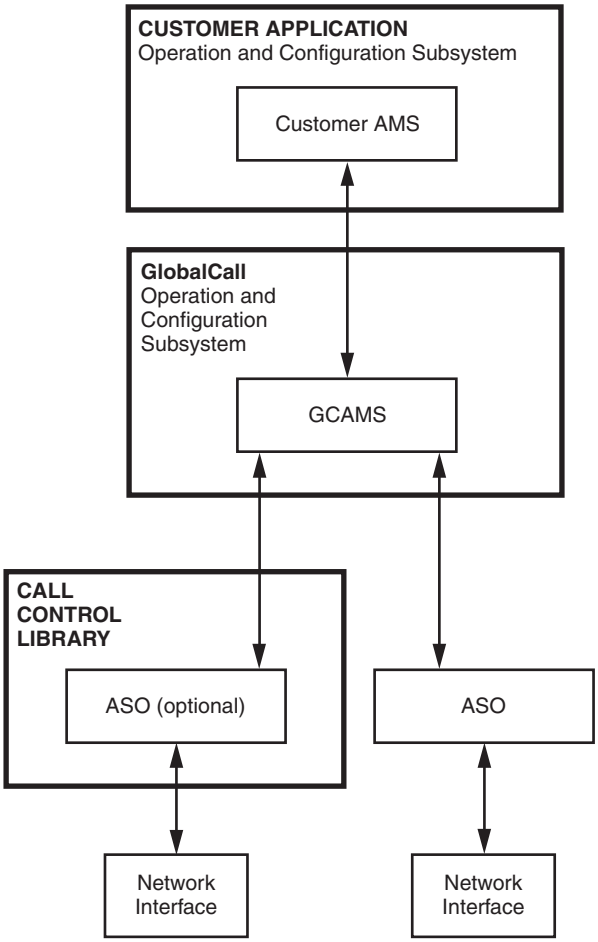
The following sections describe the components and operation of GCAMS.

8.1.1 Alarm Management System Components

The alarm management system is made up of several components, including GCAMS. The other components are the customer application's alarm management system (AMS), and the alarm source objects (ASOs). ASOs can either reside within a call control library (cclib) or separate from

a call control library. Figure 14 illustrates the relationship between the alarm management system components.

Figure 14. Architectural Diagram of Alarm Management Components



The customer application is responsible for configuring the behavior of GCAMS, including designating which alarms are blocking, which alarms the application wants to be notified of, and controlling the flow of alarms to the application. For more information, see Section 8.2.3, "Configuration of Alarm Properties and Characteristics", on page 77.

GCAMS acts as an interface between the customer application and the alarm source objects. GCAMS passes requests from the application to the ASOs, processes application configuration requests, and processes ASO alarm events. GCAMS also maintains a database of the current configuration attributes by alarm source object and line device. In addition, GCAMS implements the ASOs that are common across multiple technologies. For more on the operation and configuration of GCAMS, see Section 8.2, "Operation and Configuration of GCAMS", on page 75.

The final components of the alarm management system are the ASOs. ASOs are responsible for generating alarm events when alarms occur and then clear. If configured to do so, ASOs are also

responsible for starting and stopping the transmission of alarms and setting and getting alarm parameters, such as timing parameters.

8.2 Operation and Configuration of GCAMS

The primary functions of GCAMS are as follows:

- Generation of Events for Blocking Alarms
- Generation of Alarm Events
- Configuration of Alarm Properties and Characteristics
- Retrieving Alarm Data

8.2.1 Generation of Events for Blocking Alarms

Global Call alarms are classified as either blocking or non-blocking. Blocking alarms are alarms that cause the application to become blocked and potentially generate a GCEV_BLOCKED event when the alarm is set (the "alarm on" condition is detected). Subsequently, all blocking alarms generate a GCEV_UNBLOCKED event when the alarm clears (the "alarm off" condition is detected). Non-blocking alarms are alarms that do not cause the application to become blocked and do not generate a GCEV_BLOCKED or GCEV_UNBLOCKED event when the alarm is set or clears.

Note: The **gc_SetAlarmConfiguration( )** function can be used to change which alarms are blocking and which alarms are not blocking for a given alarm source object. To retrieve the status of the current alarm configuration, use **gc_GetAlarmConfiguration( )**. For more on changing the configuration of alarm source objects, see Section 8.2.3, "Configuration of Alarm Properties and Characteristics", on page 77.

The GCEV_BLOCKED and GCEV_UNBLOCKED events are unsolicited events that are sent in addition to other Global Call events. The blocked and unblocked events do not require any application-initiated action. The blocked event is generated only for the first blocking condition detected. Subsequent blocking conditions on the same line device will not generate additional blocked events. Until all blocking conditions are cleared, the line device affected by the blocking condition (that is, the line device that received the GCEV_BLOCKED event) cannot generate or accept calls. When the line device has completely recovered from the blocking condition a GCEV_UNBLOCKED event is sent.

When a blocking condition occurs while a call is in progress or connected, any calls on the line device that is in the blocked condition are treated in the same manner as if a remote disconnection occurred: an unsolicited GCEV_DISCONNECTED event is sent to the application and the call changes to the Disconnected state. The result value retrieved for the event will indicate the reason for the disconnection, for example, an alarm condition occurred. Result values are retrieved by calling the **gc_ResultInfo( )** function, see Section 4.4, "Event Retrieval", on page 59. The GCEV_BLOCKED event is also sent to the application to indicate that a blocking condition occurred; the **gc_ResultInfo( )** function can be called to retrieve the reason for the GCEV_BLOCKED event, as well.

The GCEV_BLOCKED and GCEV_DISCONNECTED events may arrive in any order. When the blocking condition(s) clears, an unsolicited GCEV_UNBLOCKED event is sent to the application indicating complete recovery from the blocking condition.

When a blocking condition occurs while a line device is in the Null, Disconnected, or Idle state, only the GCEV_BLOCKED event is sent since there is no call to disconnect. The call state does not change when a GCEV_BLOCKED or GCEV_UNBLOCKED event is sent to the application.

Note: In the asynchronous mode, if a **gc_WaitCall( )** function is pending when a GCEV_UNBLOCKED event is generated, the **gc_WaitCall( )** function does not need to be reissued.

The GCEV_BLOCKED and GCEV_UNBLOCKED events are generated for blocking alarms at the logical or virtual board level and the channel level:

Logical or Virtual Board Level

When the Global Call API recognizes a blocking *alarm on* condition at the logical or virtual board level, a GCEV_BLOCKED event is generated for the logical or virtual board device, assuming that the device is open. A GCEV_BLOCKED event is also generated for all time slots currently open on the logical or virtual board device, assuming that the application is currently *unblocked*. The application will receive a GCEV_BLOCKED event only for the first *alarm on* condition for a particular line device.

When the Global Call API recognizes a blocking *alarm off* condition at the logical or virtual board level, a GCEV_UNBLOCKED event is generated for the logical or virtual board device, assuming that the device is open. A GCEV_UNBLOCKED event is also generated for all time slots currently open on the logical or virtual board device, assuming there are no other blocking conditions on the line device. The application will receive a GCEV_UNBLOCKED event only for the last “alarm off” condition for a particular line device.

Channel Level

When the Global Call API recognizes a blocking *alarm on* condition at the channel level, a GCEV_BLOCKED event is generated for the channel, assuming that the application is currently *unblocked*. The application will receive a GCEV_BLOCKED event only for the first *alarm on* condition for the line device.

When the Global Call API recognizes a blocking *alarm off* condition at the channel level, a GCEV_UNBLOCKED event is generated for the time slot, assuming there are no other blocking conditions on the line device. The application will receive a GCEV_UNBLOCKED event only for the last *alarm off* condition for the line device.

8.2.2 Generation of Alarm Events

The GCEV_ALARM event can be generated by both blocking and non-blocking alarms. Blocking alarms are alarms that generate GCEV_BLOCKED and GCEV_UNBLOCKED events when the alarms set and clear. GCEV_ALARM events are for information purposes only and do not cause any channel state or call state changes.

In order for the GCEV_ALARM event to be returned by the application, the notify attribute for the specified alarm source object must be set to “on” via the **gc_SetAlarmConfiguration( )** function. In addition, the alarm source object must meet the alarm flow configuration requirements, which are set using the **gc_SetAlarmFlow( )** function or the **gc_NotifyAll( )** function. (See Section 8.2.3, “Configuration of Alarm Properties and Characteristics”, on page 77 for more information.)

When the application returns a GCEV_ALARM event, indicating that an alarm has been received, information about the alarm can be retrieved using the **gc_AlarmName( )** function. The **gc_AlarmName( )** function converts the alarm to its English name to allow for interpretation of the reason for the alarm. For more information on retrieving alarm data for a given ALARM_EVENT, see Section 8.2.4, "Retrieving Alarm Data", on page 80.

Some of the ways the information provided by the GCEV_ALARM events can be used are:

- Administration of alarms (using alarm information to determine the appropriate configuration of GCAMS)
- Detection and transmission of alarm conditions between networks (drop and insert applications)
- Manual handling of alarms for drop and insert applications
- Generating reports
- Troubleshooting connections and protocols

8.2.3 Configuration of Alarm Properties and Characteristics

GCAMS provides the ability to set the alarm configuration for line devices and alarm source objects. The initialization of ASO configuration values is done at build time.

The Global Call API provides several functions that are used to configure how, when and which alarms are sent to the application, and to define the characteristics of the alarms. These functions are:

- **gc_SetAlarmConfiguration( )**
- **gc_SetAlarmFlow( )**
- **gc_SetAlarmNotifyAll( )**
- **gc_SetAlarmParm( )**

Corresponding functions allow for the retrieval of the current status of the configurations. These functions are:

- **gc_GetAlarmConfiguration( )**
- **gc_GetAlarmFlow( )**
- **gc_GetAlarmParm( )**

The use of these functions is described in the following sections. Alarm configuration tips are also provided. For more information about the alarm configuration functions, see the *Global Call API Library Reference*.

For line devices opened by technologies that use GCAMS, there is an entity called the *network ASO ID* that is the alarm source object associated with the network. As a programming convenience, Global Call defines ALARM_SOURCE_ID_NETWORK_ID that corresponds to the network ASO ID. This define is useful in many contexts. For example, notification of all alarms on a line device can be configured using the call:

```
gc_SetAlarmNotifyAll(..., ALARM_SOURCE_ID_NETWORK_ID, ...)
```

The ALARM_SOURCE_ID_NETWORK_ID is a value that can be used to represent, for a given line device, whatever the network ASO ID happens to be.

8.2.3.1 Configuring Alarm Notification

In order for an alarm to be sent to the application, the "notify" attribute of the alarm must be set to "yes". Initially, the notify attribute of all alarms is set to "no". The **gc_SetAlarmConfiguration( )** function is used to set and change the notify attribute for a specified alarm source object on a given line device. To retrieve the status of the alarm configuration parameters, use the **gc_GetAlarmConfiguration( )** function.

Alternatively, the **gc_SetAlarmNotifyAll( )** function can be used as a shortcut when the application wants to change the notification status, that is, when the application wants to change from "notify" to "no notify", for all line devices that have the specified alarm source object.

8.2.3.2 Configuring Alarm Flow

The **gc_SetAlarmFlow( )** function is used to further refine which of the alarms are sent (that is, allowed to "flow") to the application. Alarm flow configuration is controlled on a line device basis. The alarm flow can be configured in any of the following ways:

- All alarms are sent to the application
- All, and only, blocking alarms are sent to the application
- Only the first alarm on and the last alarm off are sent to the application
- Only the first blocking alarm on and the last blocking alarm off are sent to the application

Note: To configure the alarm flow so that no alarms are sent to the application, use the **gc_SetAlarmConfiguration( )** function and set the notify attribute of all alarms to "no".

To determine the current alarm flow options, use the **gc_GetAlarmFlow( )** function.

8.2.3.3 Configuring Blocking and Non-Blocking Alarm Classification

For any given alarm source object, the **gc_SetAlarmConfiguration( )** function can be used to set and change which alarms are blocking or non-blocking. This information is stored in the ALARM_LIST data structure.

To retrieve the status of the current alarm configuration, use the **gc_GetAlarmConfiguration( )** function.

8.2.3.4 Configuring Alarm Parameters

The **gc_SetAlarmParm( )** function is used to set alarm parameters that control ASO parameters such as timing. An example of a timing parameter would be setting how long a loss of synchronization must be present before the ASO declares a loss of sync alarm or alarm handling mode.

Use of the **gc_SetAlarmParm( )** function, as well as the **gc_GetAlarmParm( )** function, is highly alarm source object dependent and requires detailed knowledge of the underlying ASO technology

by the application writer. For a description of ASOs that are common across multiple technologies, see the *Global Call API Library Reference*.

8.2.3.5 Alarm Configuration Tips

The procedures for configuring alarms depends on whether the application writer is configuring the behavior of alarm source objects or specific line devices associated with a given alarm source object. (When a line device is opened, it takes the blocking and notify attributes of the network ASO, if any, associated with the given line device.)

The default configuration (that is, the flow, blocking and notify attributes) of an alarm source object can be changed by using the **gc_SetAlarmFlow( )** and **gc_SetAlarmConfiguration( )** functions. Typically, the default configuration should be changed immediately after calling **gc_Start( )** and prior to calling **gc_OpenEx( )**.

To change the default configuration for all known ASOs, perform the following steps:

- convert the ASO name to the ASO ID using the **gc_AlarmSourceObjectNameToID( )** function
- change the attributes of the specified ASO name using the **gc_SetAlarmConfiguration( )** function

Note: Changing the attributes of an ASO requires detailed knowledge of the given ASO.

The procedures for changing the configuration of line devices depends on whether all the line devices associated with the same ASO are to have the same attributes, or if the application requires different behaviors for line devices associated with the same ASO. For those applications that require all line devices to have the same attributes, use the procedures for changing the default configuration for ASOs as described above. For applications that are intended to be cross-technology and/or more robust, the following steps should be performed to change the attributes:

1. Call **gc_OpenEx( )**.
2. Retrieve the network ASO ID associated with the line device using the **gc_GetAlarmSourceObjectIDToName( )**.
3. Convert the network ASO ID to a name using the **gc_AlarmSourceObjectIDToName( )**. This is a necessary step as not all ASOs will have a fixed ID.
4. Using the ASO name, change the attributes of the line device using the **gc_SetAlarmConfiguration( )** function.

   Note: Changing the attributes of an ASO for a specified line device requires detailed knowledge of the given ASO.

For applications that are using only one “known” technology, the application can use either **gc_GetAlarmSourceObjectNetworkID( )** to retrieve the network ASO ID associated with the line device or **gc_AlarmSourceObjectNameToID( )** to retrieve the ID for the “known” ASO.

8.2.4 Retrieving Alarm Data

The GCAMS database contains the following information:

- A list, by call control library, of all the boards that are currently open
- Information about each opened board, including the board name, the call control library ID, all open time slots on the board, alarm source objects associated with the device, and the alarm callback procedure
- A list of registered alarm source objects and their attributes. (Alarm source objects are registered automatically when the **gc_Start( )** function is called.)
- Default alarm source object data (provided by GCAMS)

8.2.4.1 Alarm Numbers and Names

Alarm events are identified in the database by name and number. The following functions are used to retrieve the names, numbers and IDs and to convert them from one to the other:

gc_AlarmName( )
converts the alarm name to English, for a given event. Alarm names are assigned by the developer for use in report generation.

gc_AlarmNumber( )
retrieves the alarm number, for a given event. Alarm numbers (values) are predefined for a given ASO. See the *Global Call API Library Reference* for ASOs that are common to multiple call control libraries.

gc_AlarmNumberToName( )
converts the alarm number to the English name

8.2.4.2 Alarm Source Object IDs and Names

Alarm source objects (ASOs) are identified in the GCAMS database by the ASO ID and by the ASO name. ASOs that are not part of a call control library have predefined names, as provided in the *Global Call API Library Reference*. The names of ASOs that are part of a call control library are provided in the *Global Call IP Technology Guide*.

The following functions are used to retrieve ASO names and IDs and to convert them from one to the other:

gc_AlarmSourceObjectID( )
retrieves the alarm source object ID, for a given event

gc_AlarmSourceObjectIDToName( )
converts an alarm source object ID

gc_AlarmSourceObjectName( )
retrieves the alarm source object name, for a given event

gc_AlarmSourceObjectNameToID( )
converts the alarm source object name to the alarm source object ID

Note: GCAMS uses predefined IDs for the ASOs it has implemented, however it is recommended that applications use the **gc_AlarmSourceObjectNameToID( )** function to associate the ASO name with an ID rather than using the ID directly. This allows for more flexible applications if ASOs that reside in call control libraries and have dynamically assigned IDs are added to the application.

In addition, the following functions are used to obtain additional information about the ASOs:

gc_GetAlarmSourceObjectList( )
gets all ASOs associated with a line device

gc_GetAlarmSourceObjectNetworkID( )
gets the network ID associated with a line device.

For more information on these functions, see the individual function descriptions in the *Global Call API Library Reference*.

8.3 Sample Alarm Scenarios

The following scenarios illustrate the relationship between the application, GCAMS, and the AOS and provide examples of alarm system configurations, and the sequence for transmission of alarms. The scenarios include:

- Scenario 1: Application Notified of First and Last Blocking Alarm
- Scenario 2: Default Behavior for Alarm Notification

8.3.1 Scenario 1: Application Notified of First and Last Blocking Alarm

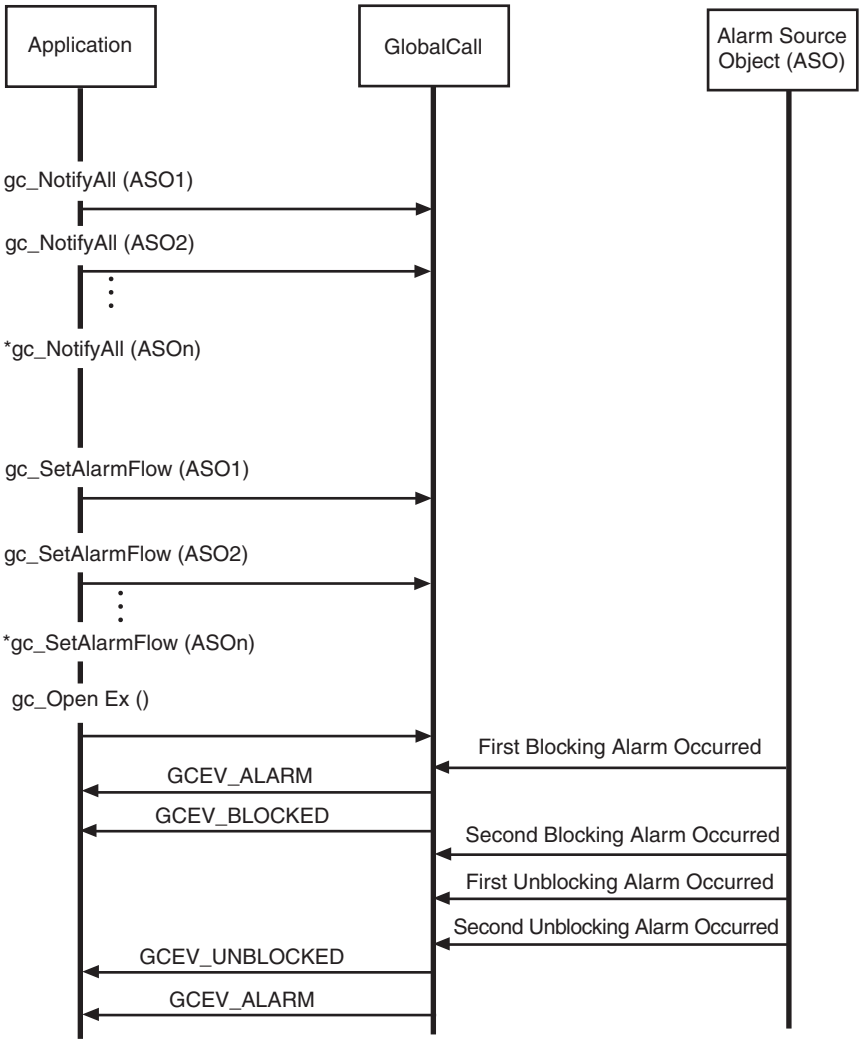
In this scenario the application wants to be notified of only the first and last blocking alarm events. The default blocking configuration is acceptable. See Figure 15.

Note: If both a GCEV_ALARM and a GCEV_BLOCKED (or GCEV_UNBLOCKED) event are generated for an alarm, the order in which these events are sent to the application is not guaranteed.

The steps are:

1. Configure all known call control libraries – set all alarms to notify and set flow control to first and last blocking.
2. Open a line device. The line device's configuration will be "inherited" from its network ASO, which has already been initialized.

Figure 15. Notification of First and Last Blocking Alarm



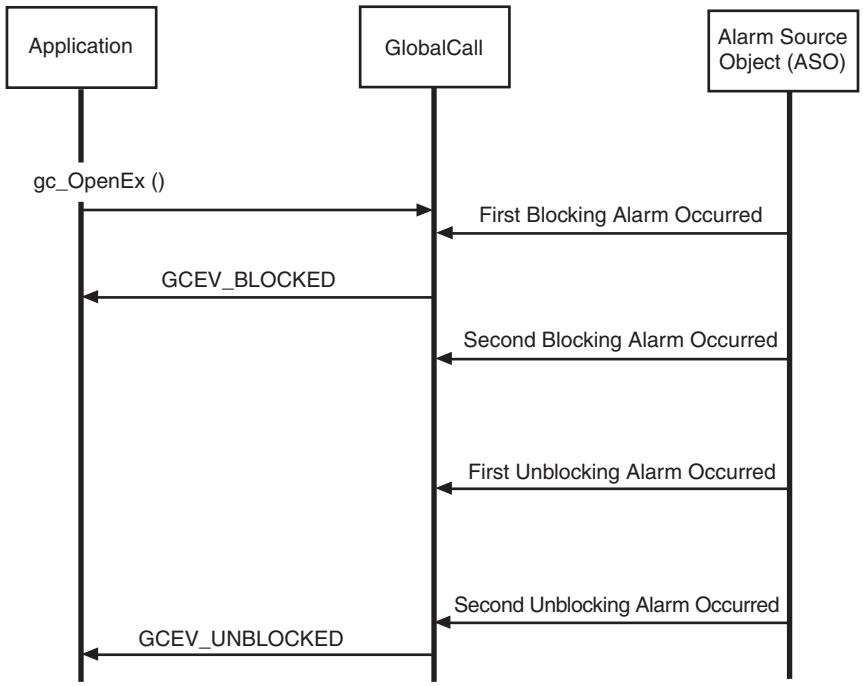
Note: * indicates that the function should be repeated for all ASO's

Note: The function calls for alarm processing are not shown.

8.3.2 Scenario 2: Default Behavior for Alarm Notification

The default behavior is that the application is not notified of alarm events. See Figure 16.

Figure 16. Default Behavior for Alarm Notification



Real Time Configuration Management 9

This chapter describes the Global Call Real Time Configuration Manager (RTCM). Topics include the following:

9.1 Real Time Configuration Manager Overview

The Global Call Real Time Configuration Management (RTCM) system manages run time configuration for Global Call components. The RTCM feature is used when the application needs to modify configuration data. If the configuration data is not modified, the application uses the initial values for the configuration.

The Global Call RTCM system allows applications to:

- Set the configuration of a protocol dynamically. For example, the default values of CDP parameters can be retrieved or updated with new values.
- Set the configuration of a physical or logical entity dynamically. The entity can be a system (that is, all boards), board, network interface, channel, or call.
- Set the configuration of a call control library dynamically. For example, the default call state mask value of a channel can be retrieved or updated with a new value.

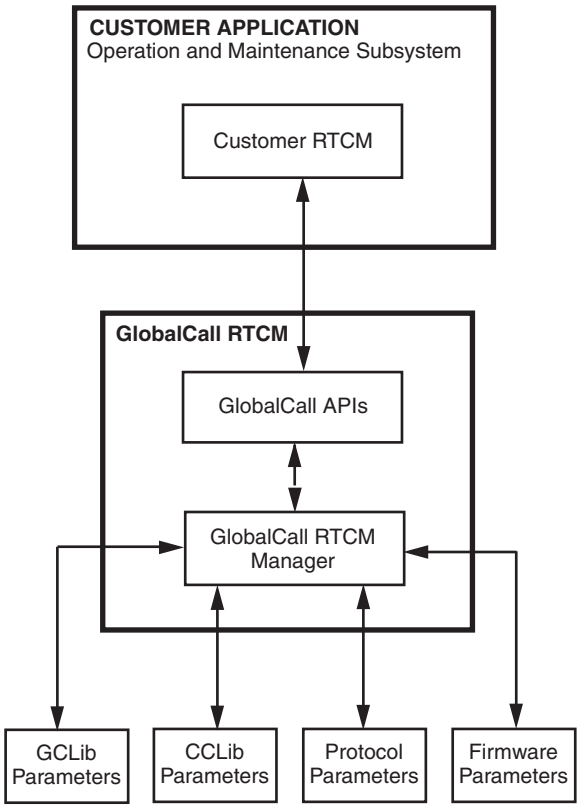
In addition, the RTCM feature provides Global Call applications with the ability to retrieve configuration parameter information. Some of the ways this information can be used include:

- Efficient network management
- Troubleshooting software and hardware
- Performance tuning
- Dynamic alteration of a target object's behavior based upon past behavior
- Generation of status reports
- Dynamic configuration of Global Call call modules or call events

9.2 RTCM Components

The RTCM comprises three major components: the customer application using RTCM, the Global Call RTCM, which consists of the Global Call RTCM API and the Global Call RTCM Manager, and the RTCM parameters. Figure 17 shows the relationship between these components.

Figure 17. Relationship of Customer Application, Global Call RTCM, and RTCM Parameters



Each of the components of the RTCM is described in the following sections.

9.2.1 Customer Application Using Global Call RTCM

The customer application interfaces with the Global Call RTCM Manager via Global Call RTCM API functions. The primary function of an application with regards to RTCM is the maintenance of parameter data. It is the application developer's responsibility to understand the impact on system operation before changing a parameter value. Specifically, the application developer is responsible for the following:

- Obtaining the information about run-time configuration support from the *Global Call IP Technology Guide*.
- Ensuring that the configurable parameters match the target entity and inserting parameter data in the proper data format.

- Choosing the proper Global Call RTCM API control parameters (programming mode, update condition, and timeout) to ensure the efficiency of the retrieve or update configuration process and that the application program is not blocked.
- Obtaining the configuration data from Global Call RTCM retrieval events.
- Correcting errors in input configurable parameter data based on the Global Call error messages.

9.2.2 Global Call RTCM

The Global Call RTCM acts as an interface between the customer application and the configurations of the target objects. A target object is a configurable basic entity and is represented by its target type and target ID (for more information, see Section 1.5.4, "Target Objects", on page 23).

As mentioned before, the Global Call RTCM comprises the RTCM Manager and the RTCM API functions.

The RTCM Manager is responsible for configuring components, including the Global Call Library (GCLib), Call Control Library (CCLib), protocol, and firmware parameters (see Section 9.3, "Using RTCM Parameters", on page 88).

The **gc_SetConfigData( )** function is used to set configuration parameters (consisting of a specified target object and the configuration data) from the customer application to the software module where the target object is located. The Global Call RTCM maintains the information about a target object with its associated software module so that the Global Call RTCM can call the appropriate software module to execute the configuration request. The Global Call RTCM also assigns a unique ID for each request and outputs it to the application. The ID is used by the application for tracking function calls.

Note: The **gc_SetConfigData( )** function is only supported in asynchronous mode and can operate only on a board device.

In addition, the Global Call RTCM returns an error value when the function returns in synchronous mode or generates a Global Call event related to the Global Call RTCM in asynchronous mode.

Since the Global Call RTCM may not have any knowledge about configurable parameters defined or used in individual modules, it passes the configuration request to the software module in which the target object is located. The customer application must ensure that the target object and requested parameters match.

9.2.3 RTCM Parameters

The third component of the RTCM feature are the RTCM Parameters. The parameters are defined and maintained in four categories of software modules: Global Call Library (GCLib), Call Control Library (CCLib), Protocol and Firmware. Each software module supports different target objects as well as the target objects' parameters.

9.3 Using RTCM Parameters

The Global Call RTCM provides a generic way of setting the configuration information for a target object. The target objects and their parameters are defined and maintained in the following categories of software modules:

Parameters in GCLib Module
: parameters that are defined in GCLib. These parameters are common across multiple technologies, such as protocol name and ID, call event mask, and the call state mask of a line device. Although the GCLib module maintains many of the GCLib-defined parameters, some parameters, such as **calling number** and **call info**, are maintained in other modules (such as CCLib).

Parameters in CCLib Module
: parameters that are defined and maintained in the CCLib module. The CCLib may maintain some GCLib-defined parameters, such as **calling number** and **call info**. See the *Global Call IP Technology Guide* for more information about configurable parameters.

Parameters in Protocol Module
: parameters that are defined and maintained in a protocol module. One example of protocol parameters are country dependent parameters (CDP). See the *Global Call IP Technology Guide* for more information about configurable parameters.

Parameters in Firmware Module
: parameters that are defined and maintained in a firmware module. See the *Global Call IP Technology Guide* for more information about configurable parameters.

To access the value of a parameter, the application must specify a four-part name consisting of two pairs: (target object type, target object ID) and (set ID, parameter ID).

Target object type and target object ID
: This pair represents the target object. See Section 1.5.4, "Target Objects", on page 23 for more information. Both the target object type and target object ID are specified as the first two arguments to the Global Call RTCM API function. An example of a target object is (GCTGT_CCLIB_CHAN, Global Call line device ID).

Set ID and parameter ID
: This uniquely represents a parameter within a specified target object. See Section 9.4, "Setting Parameter Information", on page 89 for more information. A set ID typically represents a group of parameters that are closely related and are maintained in the same software module. The parm ID represents a parameter within a given set ID. In general, parameter IDs are only guaranteed to be unique within a given set ID. Note that some configurable parameters are defined only for a specific software module, while others may be used across different software modules. Typically, a software module that supports RTCM contains multiple parameter sets as well as target objects.

  Note: The set ID and parm ID pairs are used by other Global Call features in addition to RTCM.

9.3.1 Parameter Dependencies

A high-level target object, such as a system entity, can contain a lower-level target object, such as a channel entity. When a target object is created, its configuration is initialized as the default or

current value, depending on its implementation. If a parameter is defined and used for both the high-level and the lower-level target object, updating the parameter of the high-level target object may also cause the same parameter of the newly-created lower-level target object to be updated. Consult the *Global Call IP Technology Guide* for information about parameter usage.

9.3.2 Parameter Definitions

GCLib or CCLib parameter descriptions can be found in the *Global Call API Library Reference*. Other target objects and their associated set IDs and parameters are described in the *Global Call IP Technology Guide*. The *Global Call IP Technology Guide* also includes which header files are required.

All configurable parameters are of the following type:

Static
: parameters that are predefined in header files with a fixed set ID and parameter ID

Every parameter is further defined by the software module as one of the following update conditions:

read-only
: parameter is not allowed to be changed by the application

update immediately
: parameter is updated immediately upon a set request

update-at-null call state
: parameter is only allowed to be updated at the Null call state (that is, there are no active calls). This parameter is updated after a set request is made and when the call state is Null.

See Section 9.4, “Setting Parameter Information”, on page 89 and the *Global Call IP Technology Guide* for detailed information.

9.4 Setting Parameter Information

The Global Call RTCM feature supports the updating of multiple parameters of the same target object in a single Global Call API function call. The function used to set configuration data is as follows:

gc_SetConfigData( )
: updates the configuration data of a given target object

The function call must include a valid target object that is consistent with the target ID. In addition, the following conditions must exist:

- Valid parameters (set ID and parm ID) supported by this target object
- Correct parameter data type and data value
- Appropriate control parameters (programming mode, timeout, update condition) have been set.

The set ID and parm ID as well as the data type and data value are specified in the function call using the GC_PARM_BLK data structure.

9.4.1 GC_PARM_BLK Data Structure

As an argument of the **gc_SetConfigData( )** function, the configuration data is required to be a generic GC_PARM_BLK data structure. The Global Call application must input parameter information, such as the set ID, parm ID, and value, strictly following entry specifications. In addition to inputting a valid set ID and parameter ID, the parameter value size must match the parameter data type. For example, a *long* data type has four bytes. A character string value is terminated by a NULL (\0). The Global Call utility functions must be used to allocate or deallocate the GC_PARM_BLK memory, insert a parameter, or retrieve a parameter. See the *Global Call API Library Reference* for more information on the utility functions (gc_util_xxx functions).

The customer application should not configure the same parameter more than once in one single function call; otherwise, the results will be undetermined. Also, the customer application must only configure one target object in one function call. Otherwise, the mixture of parameters of different target objects in the GC_PARM_BLK will be rejected by the Global Call RTCM API function.

9.4.2 Control Parameters

The Global Call RTCM API control parameters ensure the efficiency of the update configuration process and that the application program is not blocked. The application can specify:

- the programming mode
- the timeout interval for completing the update
- the update condition; that is, whether the update should occur either at the Null call state or immediately when updating the parameters of a target object with an active call.

9.4.2.1 Programming Mode

The customer application can specify whether to access configurations in the asynchronous mode or synchronous mode. The following describe how the **gc_SetConfigData( )** function operates in the asynchronous and synchronous programming modes:

gc_SetConfigData( )

Synchronous Mode: Upon completion of the function call, the **gc_SetConfigData( )** function returns a value of GC_SUCCESS to indicate that all requested parameters in a given target object have been successfully updated. Any other return value indicates that at least one requested parameter in a target object failed to be updated due to an error. The **gc_ErrorInfo( )** function is called immediately to obtain the last error and additional message describing the parameter and the error (pointer to the additional message field). During the **gc_SetConfigData( )** function call, once an error occurs, Global Call stops updating the remaining parameters and returns an error value to the application. If this function call requires updating multiple parameters in a target object, the parameters before the error may have been updated while other parameters will not have a chance to be updated.

gc_SetConfigData( )

Asynchronous Mode: The Global Call application receives the GCEV_SETCONFIGDATA event if all the requested parameters in a given target object are successfully updated. Otherwise, the Global Call application receives the GCEV_SETCONFIGDATA_FAIL event, which indicates that at least one requested parameter in the target object failed to update due to an error. The METAEVENT data structure, which is associated with both events, has a field, evtdatap, that points to a GC_RTCM_EVTDATA data structure. The GC_RTCM_EVTDATA data structure provides the error value and additional message describing the parameter and the error.

The original GC_PARM_BLK data block is not changed after the **gc_SetConfigData( )** function returns.

9.4.2.2 Timeout Option

The timeout option provided by the timeout parameter in the **gc_SetConfigData( )** function is not supported and should be set to 0.

9.4.2.3 Update Condition

When using the **gc_SetConfigData( )** to update the parameters of a target object with an active call, the application can specify whether the update should occur either at the Null call state or immediately. If parameters are to be updated at the Null state, but the function requests to immediately update them while the target object has any active call, the function returns an error to the application. If parameters are to be updated immediately, the function can update them immediately or at the Null state.

Table 11 describes the possible settings and resulting actions for the update condition as used by the **gc_SetConfigData( )** function.

Table 11. Update Condition Flag and Global Call Process

Update condition flag (Global Call APP)	Parameter Update Allowed in Target Object	Target Object Status	Global Call Action
GCUPDATE_IMMEDIATE	Update immediately	Active or no active call	Update parameter
	Update at Null state	No active call	Update parameter
		Active call	Return error
GCUPDATE_ATNULL	Update immediately	No active call	Update parameter
		Active call	Postpone until no active call
	Update at Null state	No active call	Update parameter
		Active call	Postpone until no active call

The **gc_ResetLineDev( )** function is used to speed the update of the parameters that are waiting for the arrival of the Null state. For example, the customer application can call the **gc_SetConfigData( )** function multiple times to request the parameters to be updated at the Null

state. Instead of waiting for the Null state, the customer application can call the **gc_ResetLineDev( )** function to reset the channel to the Null state and update all the parameters.

9.5 Handling RTCM Errors

Configuration data for multiple parameters of a target object can be updated in a single function call. The function will abort on any single parameter retrieval failure. If the function returns a Global Call error, the application calls the **gc_ErrorInfo( )** function immediately to obtain the last error code, error message, and additional message. An additional message identifies which parameter has an error. In the asynchronous mode, the application calls the **gc_ResultInfo( )** function immediately to obtain the result value, error message, and additional message.

See the *Global Call API Library Reference* for Global Call RTCM error values and messages.

9.6 Configuration Procedure

The basic steps for using the Global Call RTCM feature are:

1. Ensure that the target object has been opened or loaded and find the target object ID.
2. Find the parameter information (set ID, parm ID, and data type) related to the target object.
3. Find the parameter update condition or requirement. Understand the impact on the operation of itself or other target objects after change of parameters.
4. Select the appropriate programming mode, timeout, and update condition (if applicable) to allow Global Call to finish the request efficiently without blocking the application program.

9.7 Sample Scenario Using gc_SetConfigData( )

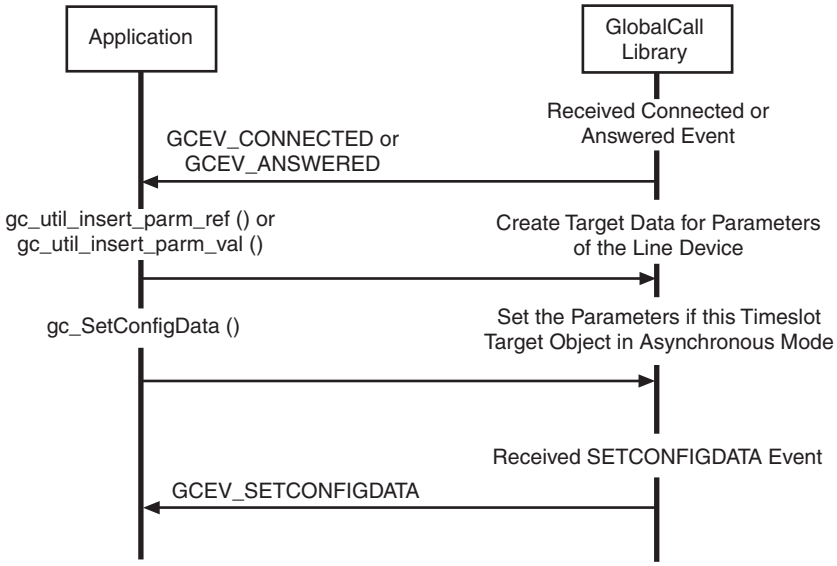
The following is an example of how a customer application can use the Global Call **gc_SetConfigData( )** in asynchronous mode to set the configuration of various target objects.

Asynchronous mode is generally suggested to be used in either of the following cases:

- The request is to update parameters that are call related and the channel is not at the NULL state
- The **target type** is GCTGT_CCLIB_NETIF.

Figure 18 shows the procedure for setting line device configuration in asynchronous mode.

Figure 18. Setting Line Device Configuration in Asynchronous Mode



The procedure for setting the configuration of a Line Device in asynchronous mode is as follows:

1. The channel has an active call. Create the target object data (that is, a GC_PARM_BLK data structure) with the appropriate **set ID**, **parm ID**, **value size**, and **value buffer** by calling the Global Call utility functions. See the *Global Call API Library Reference* for more information.
2. Call the **gc_SetConfigData( )** function with:
   target_type = GCTGT_CCLIB_NETIF
   target_id = Global Call line device ID
   time_out = 0
   mode = EV_ASYNC
   update condition = GCUPATE_ATNULL
3. If the **gc_SetConfigData( )** function is successful, a GCEV_SETCONFIGDATA event is received. If the GCEV_SETCONFIGDATA_FAIL event is received, call the **gc_ResultInfo( )** function to find the error and correct it.

Handling Service Requests 10

This chapter describes the Global Call Service Request (GCSR) feature. Topics include the following:

10.1 Service Request Overview

The Global Call Service Request (GCSR) feature is an optional feature that allows a device to send a request to another remote device for some kind of service. Some examples of the services that may be requested are:

- Device Registration
- Channel Setup
- Call Setup
- Information Requests
- Operational Requests

In general, this feature is useful when a Global Call application needs to make a request between two Global Call devices across a network.

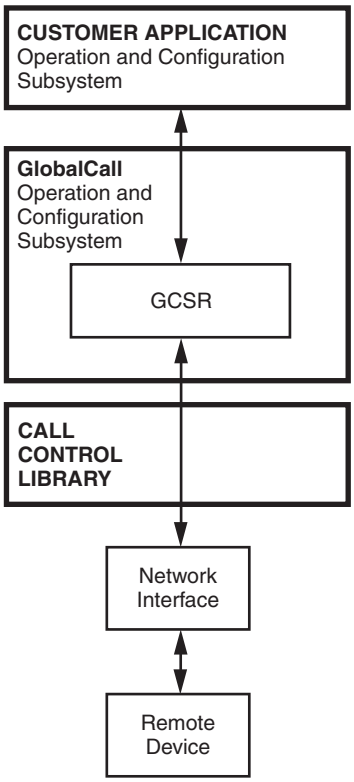
Some examples of typical uses are:

- Registration Requests
- Administration Requests (for example, logon requests)
- Bandwidth Requests
- Capabilities Requests (for example, determining remote-side capabilities)
- Preference Requests (for example, informing remote-side of setup preferences)

Since this feature is rather generic, the capabilities for IP technology are largely dependent on the support provided by the call control libraries for that technology. Refer to the *Global Call IP Technology Guide* for more information.

Figure 19 shows the architecture of the GCSR feature.

Figure 19. Service Request Architecture



10.2 Service Request Components

Using the Global Call Service Request (GCSR) feature involves the following API components:

gc_ReqService( )
function to make a request

gc_RespService( )
function to respond to a request

GCEV_SERVREQ
an event indicating that a request has been received

GCEV_SERVRESP
an event indicating a response has been received; therefore, this is also a termination event for the **gc_ReqService( )** function

GCEV_SERVRESPCMPLT
termination event for the **gc_RespService( )** function

When using the GCSR, all requests and responses are to be made on specific device targets (that is, LDID, CRN), and depending on the type of request and the call control library used, additional restrictions may apply. See the *Global Call IP Technology Guide* for more information.

10.3 Service Request Data

All information transmitted and received using the Service Request feature is done using the generic GC_PARM_BLK data structure. Three parameter IDs, under the GCSET_SERVREQ set ID, are used for all requests and responses:

PARM_SERVICEID (unsigned long)
the service identification number. This is a number assigned by the call control library to distinguish between requests and is used as follows:

- When making a request (**gc_ReqService( )**), ignore this field.
- When generating a response (**gc_RespService( )**), this value needs to be set to the same ID as the ID of the received request (through GCEV_SERVREQ).
- When receiving a response (through GCEV_SERVRESP), this field should match the ID assigned when the request was first made.

PARM_REQTYPE (int)
the type of request made. Refer to the *Global Call IP Technology Guide* for the actual values.

PARM_ACK (short)
the acknowledgement field with the following usage:

- When used for a service request, a value of GC_ACK indicates that a response is required, and a value of GC_NACK indicates that no response is necessary.
- When used for a service response, a value of GC_ACK indicates a confirmation, and a value of GC_NACK indicates a rejection.

Depending on the call control library used, additional parameters may also be used. Refer to the *Global Call API Library Reference* for more information about the GC_PARM_BLK data structure. Before the Service Request feature can be used, a GC_PARM_BLK data structure must be set up to handle the data associated with the service request. Each request or response is assigned a Service ID by the call control library and should be used by the application when generating responses as well as to distinguish among different request and responses. See the GC_PARM_BLK data structure and utility functions (gc_util_xxx) in the *Global Call API Library Reference* for more information on setting up the data structure for the Service Request feature.

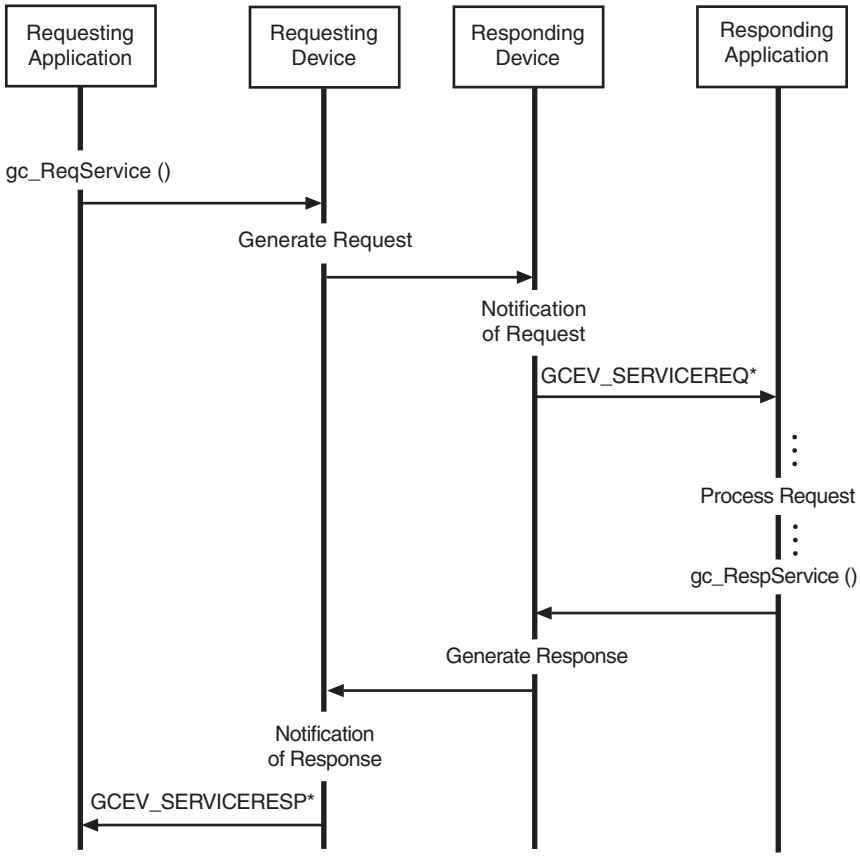
Notes: ***1.*** When using the **gc_ReqService( )** function, PARM_REQTYPE and PARM_ACK are *mandatory* parameters of the GC_PARM_BLK pointed to by the **reqdatap** function parameter.

2. When using the **gc_RespService( )** function, PARM_SERVICEID is a *mandatory* parameter of the GC_PARM_BLK pointed to by the **datap** function parameter.

10.4 General Service Request Scenario

Figure 20 is a general scenario of how the Service Request feature operates in asynchronous mode. Since the Service Request feature is generic, the nature of each request and response depends on the underlying call control library. Refer to the *Global Call IP Technology Guide* for more information.

Figure 20. Generic Service Request Operation



Note: * Indicates that the extdatap of each of these events contains a pointer to GC_PARM_BLK, which in turn contains all the information associated with the corresponding request or response. The pointer is only valid until the next call to gc_GetMetaEvent () or gc_GetMetaEventEx ().

Using Global Call to Implement Call Transfer 11

This chapter describes how call transfer is implemented using Global Call. The information in this chapter is technology independent. For more specific information about implementing call transfer on IP technology, see the *Global Call IP for Host Media Processing Technology Guide*. The topics discussed in this chapter are:

11.1 Introduction to Call Transfer

Global Call supports the following call transfer methods:

- Blind Call Transfer
- Supervised Call Transfer

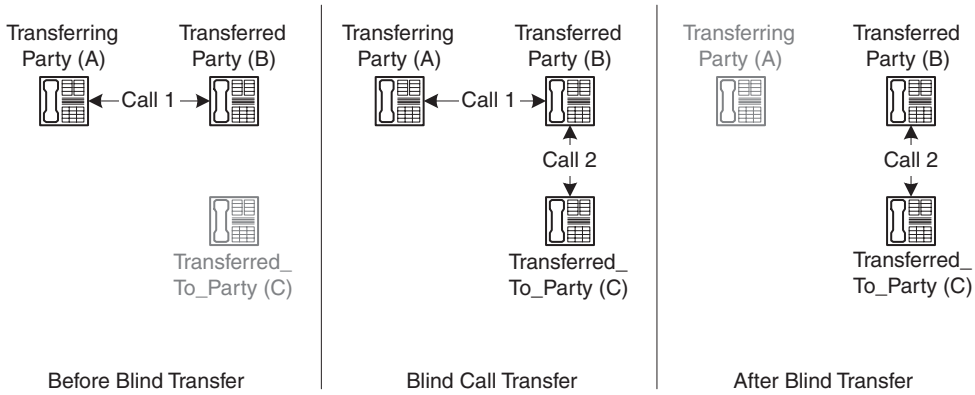
11.1.1 Blind Call Transfer

In a blind call transfer scenario, party A transfers the call between A and B (Call 1) to a call between party B and C without consulting party C. Party A places the primary call (Call 1) on hold, directly dials the Party C address, and then disconnects from Call 1 before the second call (Transferred-To Call, Call 2 - between B and C) is established. Party A may also request party B to dial Party C's address and then disconnect from Call 1 after Call 2 between B and C has been established.

Before call transfer can occur, party A must be in a call with party B (Primary Call, Call 1).

Note: In the scenario shown in Figure 21, party B initiates the transferred call to party C.

Figure 21. Blind Call Transfer (Unsupervised Transfer)

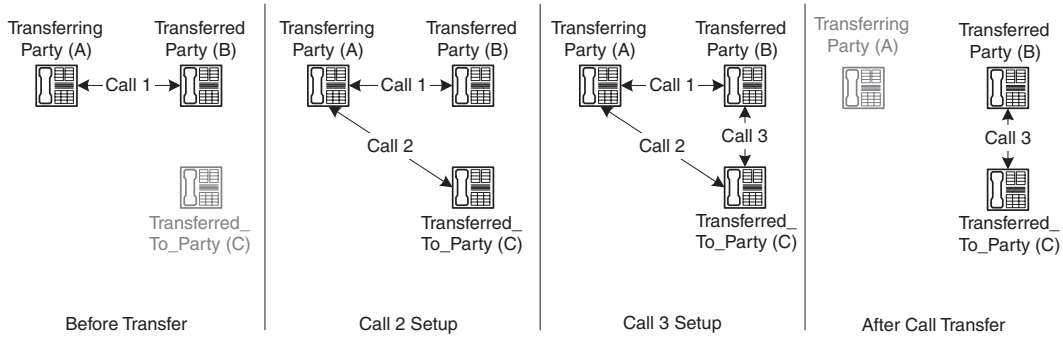


11.1.2 Supervised Call Transfer

In a supervised call transfer scenario, party A transfers the call between A and B (Call 1) to a call between party B and C after establishing a consultation call with party C. In this call, party A informs Transferred-to party C of the intent of transferring party B to party C and collects the feedback and rerouting address from party C. Party A requests that party B dial party C's address (rerouting address) and then disconnect from Calls 1 and 2 after Call 3 (Transferred-to call) between party B and party C has been established.

Before call transfer can occur, party A must be in a call with party B (Primary Call, Call 1).

Figure 22. Supervised Call Transfer



11.2 Call Transfer State Machine

Table 12 lists the new Global Call call states for blind call transfer.

Table 12. New Global Call Transfer Call States

New Call State	Description	Trigger Event
GCST_INVOKE_XFER_ ACCEPTED	The transfer request has been accepted by the remote party	GCEV_INVOKE_XFER_ACCEPTED (unsolicited event)
GCST_INVOKE_XFER	The invoke transfer is successful (i.e., the transfer is completed at transferring party)	GCEV_INVOKE_XFER (termination event for the **gc_InvokeXfer( )** function)
GCST_REQ_XFER	Receive a transfer request and wait for accept/reject	GCEV_ REQ_XFER (unsolicited event)
GCST_ACCEPT_XFER	Accepted the transfer request	GCEV_ACCEPT_XFER (termination event for the **gc_AcceptXfer( )** function)
GCST_XFER_CMPLT	Transfer is completed at transferred party	GCEV_XFER_CMPLT (unsolicited event)
GCST_REQ_INIT_XFER	Receive a transfer initiate request and wait for accept/reject	GCEV_ REQ_INIT_XFER (unsolicited event)

Note: The state diagrams in Figure 23 and Figure 24 apply to the case where party B initiates the transferred call to party C (see Figure 21), and **not** to the case where Party A places the primary call with party B on hold and then calls party C.

Figure 23. Call State Model for Blind Call Transfer at Party A

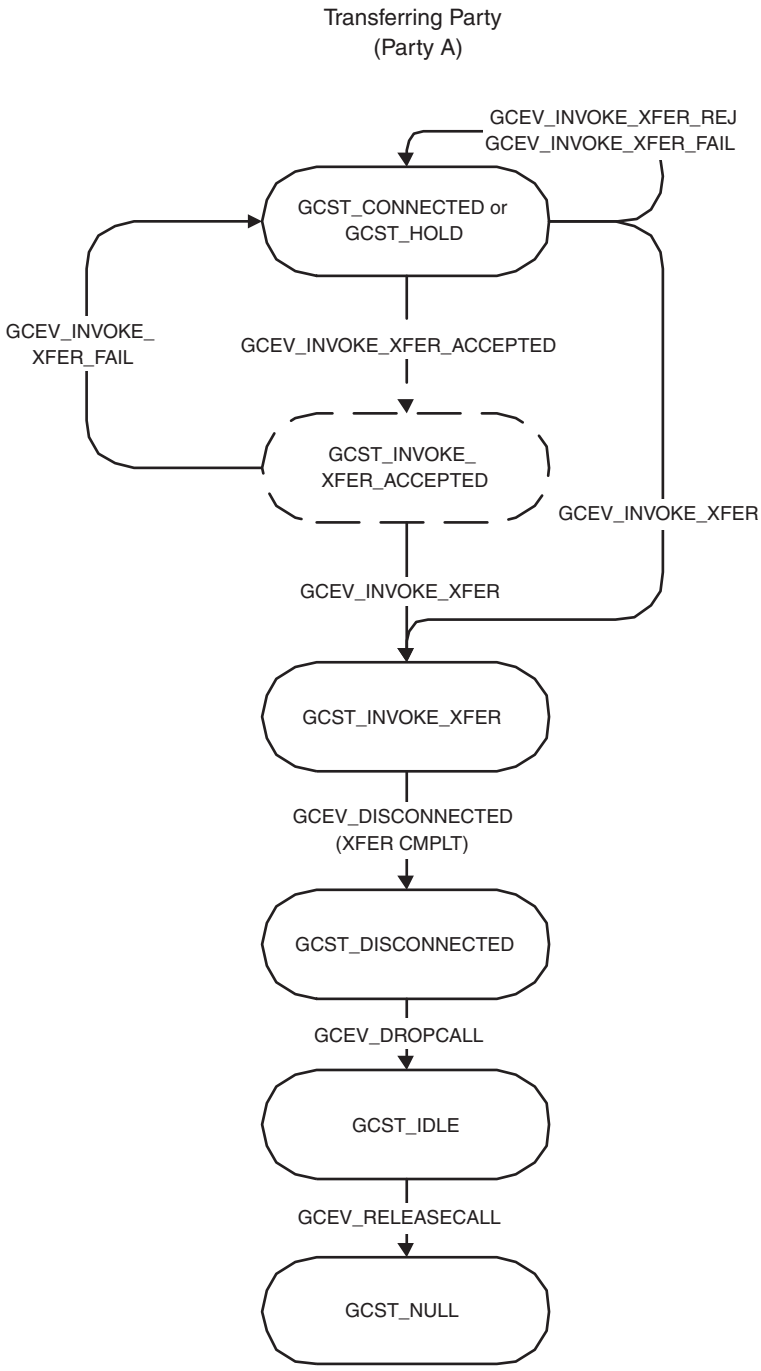
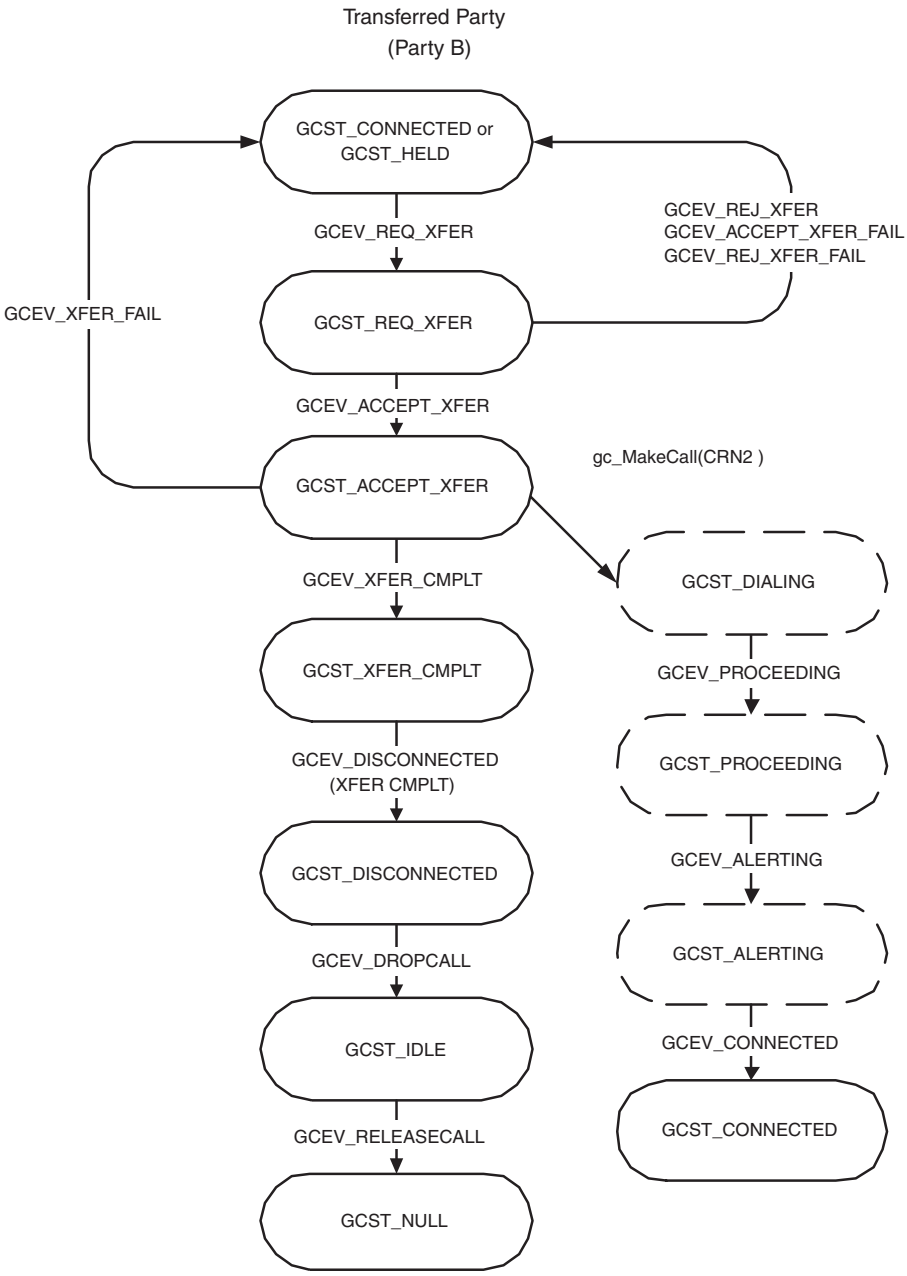


Figure 24. Call State Model for Blind Transfer at Party B



Transferred-to Party (Party C) - the rerouting call is same as new incoming call, except GCEV_DETECTGED / GCEV_OFFERED with a flag indicating a transfer call

Note: The state diagrams in Figure 25, Figure 26 and Figure 27 apply to the supervised transfer case represented in Figure 22.

Figure 25. Call State Model for Supervised Transfer at Party A

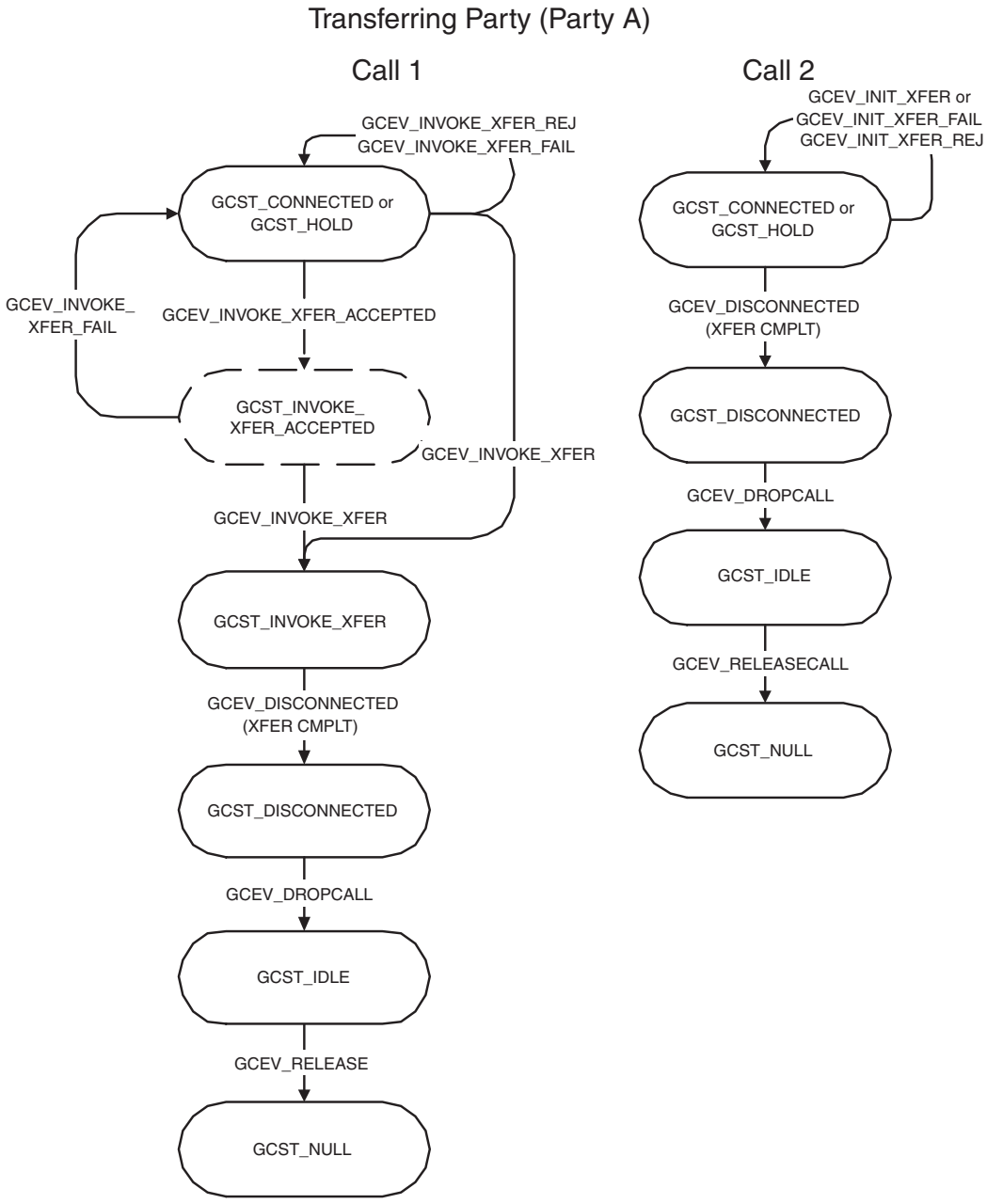


Figure 26. Call State Model for Supervised Transfer at Party B

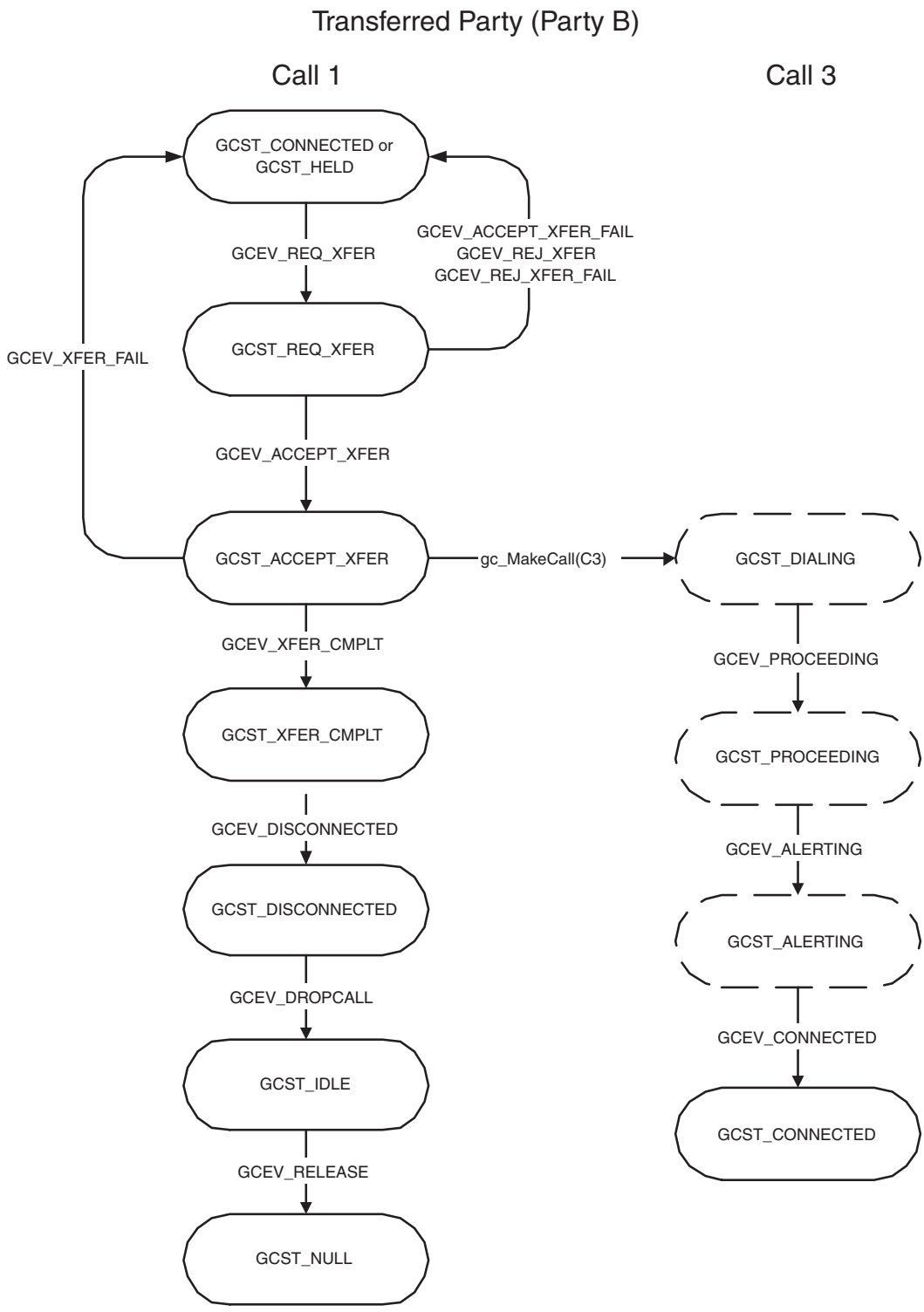
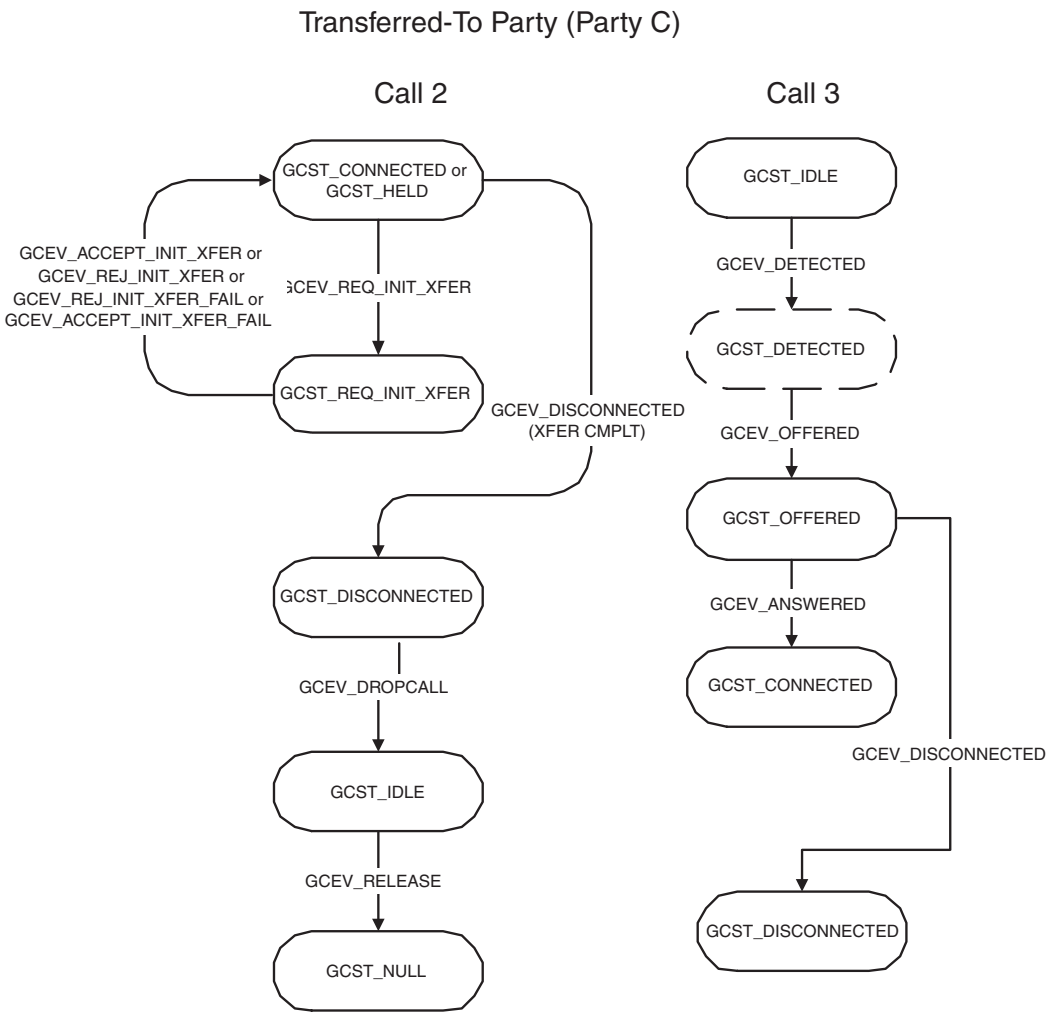


Figure 27. Call State Model for Supervised Transfer at Party C



Building Applications 12

This chapter provides general information for build applications that use the Global Call software. For additional technology-specific information, refer to the *Global Call IP Technology Guide*. Topics included in this chapter are:

12.1 Compiling and Linking

An application that uses the Global Call software must include references to the Global Call header files and must include the appropriate library files. In addition, when using specific protocols, other libraries and protocol modules are dynamically loaded. The Windows libraries may be linked and run using Microsoft Visual C++ (version 6.x or later). The following topics provide more information:

- Include Files
- Required Libraries
- Variables for Compiling and Linking Commands
- Dynamically Loaded Libraries

12.1.1 Include Files

The following header files contain equates that are required for each application that uses the Global Call library:

gclib.h
: primary Global Call header file

gcerr.h
: header file containing equates for error codes

Note: See the *Global Call IP Technology Guide* for IP-specific header files.

12.1.2 Required Libraries

The following library files must be linked to the application:

libgc.lib
: the primary Global Call library file.

libdxxmt.lib
: the primary Voice library file. This library is only required if the application uses voice library functions directly, for example, **dx_play( )**.

12.1.3 Variables for Compiling and Linking Commands

The following variables provide a standardized way of referencing the directories that contain header files and shared objects:

INTEL_DIALOGIC_INC
Variable that points to the directory where header files are stored.

INTEL_DIALOGIC_LIB
Variable that points to the directory where shared library files are stored.

These variables are automatically set at login and should be used in compiling and linking commands. The following is an example of a compiling and linking command that uses these variables:

```
cc -I${INTEL_DIALOGIC_INC} -o myapp myapp.c -L${INTEL_DIALOGIC_LIB} -lgc
```

Note: It is strongly recommended that developers use these variables when compiling and linking applications. The names of the variables will remain constant, but the values may change over time.

12.1.4 Dynamically Loaded Libraries

When the **gc_Start( )** function is called, the configured library or libraries that are used by the application are dynamically loaded. The libraries include:

libgch3r.dll
IP call control library

libgcipm.dll
IP call control library

If a configured library cannot be found, the Global Call API enters an error message in the event logger.

Debugging 13

The facilities that Global Call provides for debugging applications are IP-specific. See the *Global Call IP Technology Guide* for debugging information.

Glossary

ASO: Alarm Source Object. The source of an alarm, for example, either a physical alarm or a logical alarm.

ANI: Automatic Number Identification. A service that identifies the phone number of the calling party.

ASCII: American Standard Code for Information Interchange.

asynchronous function: A function that returns immediately to the application and returns a completion/termination at some future time. An asynchronous function allows the current thread to continue processing while the function is running.

asynchronous mode: Classification for functions that operate without blocking other functions.

available library: A call control library configured to be recognized by the Global Call API and successfully started by the Global Call gc_Start( ) function.

blind dialing: Dialing without waiting for dial tone detection.

blind transfer: See *unsupervised transfer.*

blocked: The condition of a line device initially when it is opened and after a GCEV_BLOCKED event has been received on that line device. When a line device is in a blocked condition, the application can only perform a limited subset of the Global Call commands on that line device. Call related functions may not be called with the exception of gc_DropCall( ) and gc_ReleaseCallEx( ). Non-call related functions are generally allowed. See also "unblocked" below.

blocking alarm: An alarm that causes a GCEV_BLOCKED event to be sent to the application. When the application receives a GCEV_BLOCKED event, the line device is blocked which means only a limited subset of the Global Call commands are available to the application.

call control: The process of setting up a call and call tear-down.

call control library: A collection of routines that interact directly with a network interface. These libraries are used by the Global Call functions to implement network specific commands and communications.

Call Reference Number (CRN): A number assigned by the Global Call library to identify a call on a specific line device.

call states: Call processing stages in the application.

configured library: A call control library supported by the Global Call API.

congestion: Flow of user-to-user data

CRN: See Call Reference Number.

data structure: Programming term for a data element consisting of fields, where each field may have a different definition and length. A group of data structure elements usually share a common purpose or functionality.

device handle: Numerical reference to a device, obtained when a device is opened. This handle is used for all operations on that device. See also *Call Reference Number.*

DDI string: A string of Direct Dialing In digits that identifies a called number.

DLL (Dynamically Linked Library): In Windows environments, a sequence of instructions, dynamically linked at runtime and loaded into memory when they are needed. These libraries can be shared by several processes.

device: Any computer peripheral or component that is controlled through a software device driver.

device channel: An Intel® Dialogic® data path that processes one incoming or outgoing call at a time. Compare to time slot.

driver: A software module that provides a defined interface between a program and the hardware.

en-bloc mode: Mode where the setup message contains all the information required by the network to process the call, such as the called party address information.

event: An unsolicited communication from a hardware device to an operating system, application, or driver. Events are generally attention-getting messages, allowing a process to know when a task is complete or when an external event occurs.

extended asynchronous: In Windows environments, the extended asynchronous (multithread asynchronous) model extends the features of the asynchronous model with the extended functions, **sr_WaitEvtEx( )** and **gc_GetMetaEventEx( )**. These extended functions allow an application to run different threads, wherein each thread handles the events from a different device.

failed library: A call control library configured to be recognized by the Global Call API and which did not successfully start when the Global Call gc_Start( ) function was issued.

glare: When an inbound call arrives while an outbound call is in the process of being setup, a *glare* condition occurs. Unless the protocol specifies otherwise, the incoming call takes precedence over the outbound call.

Global Call: A unified, high-level API that shields developers from the low-level signaling protocol details that differ in countries around the world. Allows the same application to easily work on multiple signaling systems worldwide (for example, ISDN, T1 robbed bit, R2/MF, pulsed, SS7, IP H.323 etc.).

IA5: International Alphabet No. 5 (defined by CCITT).

Line Device Identifier (LDID): A unique number that is assigned to a specific device or device group by Global Call.

main thread: See *thread.*

multitasking functions: Functions that allow the software to perform concurrent operations. After being initiated, multitasking functions return control to the application so that during the time it takes the function to complete, the application program can perform other operations, such as servicing a call on another line device.

multithread asynchronous: see extended asynchronous.

network handle: SRL device handle associated with a network interface board or time slot; equivalent to the device handle returned from the network library's dt_open( ) function.

network resource: Any device or group of devices that interface with the telephone network. Network resources include digital network interface devices. Network resources are assigned to telephone lines (calls) on a dedicated or a shared resource basis. Network resources control the signal handling required to manage incoming calls from the network and the outgoing calls to the network.

null: A state in which no call is assigned to the device (line or time slot).

preemptive multitasking: A form of multitasking wherein the execution of one thread or process can be suspended by the operating system to allow another thread to execute. Windows uses preemptive multitasking to support multiple simultaneous processes.

primary thread: See *thread*.

process (Windows): (1) an executing application comprising a private virtual address space, code, data and other operating system resources, such as files, pipes and synchronization objects that are visible to the process. A process contains one or more threads that run in the context of the process. (2) is the address space where the sequence of executable instructions is loaded. A process in Windows consists of blocks of code in memory loaded from executables and dynamically linked libraries (DLL). Each process has its own 4 GB address space and owns resources such as threads, files and dynamically allocated memory. Code in the address space for a process is executed by a thread. Each process comprises at least one thread which is the component that Windows actually schedules for execution. When an application is launched, Windows starts a process and a primary thread. Windows processes: 1). are implemented as objects and accessed using object services; 2). can have multiple threads executing in their address space; 3). have built-in synchronization for both process objects and thread objects. Unlike other operating systems, Windows does not use a parent/child relationship with the processes it creates.

receive: Accepting or taking digitized information transmitted by another device.

result value: Describes the reason for an event.

RFU: Reserved for future use.

SRL (Standard Runtime Library): A Intel® Dialogic library that contains C functions common to all Intel® Dialogic devices, a data structure to support application development, and a common interface for event handling.

supervised transfer: A call transfer in which the person transferring the call stays on the line, announces the call, and consults with the party to whom the call is being transferred before the transfer is completed.

synchronization objects: Windows executive objects used to synchronize the execution of one or more threads. These objects allow one thread to wait for the completion of another thread and enable the completed thread to signal its completion to any waiting thread(s). Threads in Windows are scheduled according to their priority level (31 levels are available) and run until one of the following occurs: 1) its maximum allocated execution time is exceeded, 2) a higher priority thread marked as waiting becomes waiting or 3) the running thread decides to wait for an event or an object.

termination condition: An event that causes a process to stop.

termination events: Global Call events returned to the application to terminate function calls.

thread (Windows): The executable instructions stored in the address space of a process that the operating system actually executes. All processes have at least one thread, but no thread belongs to more than one process. A multithreaded process has more than one thread that are executed seemingly simultaneously. When the last thread finishes its task, then the process terminates. The main thread is also referred to as a primary thread; both main and primary thread refer to the first thread started in a process. A thread of execution is just a synonym for thread.

tone resource: Same as a voice resource except that a tone resource cannot perform voice store and forward functions.

transmit: Sending or broadcasting of digitized information by a device.

unsolicited event: An event that occurs without prompting (for example, GCEV_BLOCKED, GCEV_UNBLOCKED, etc.).

unblocked: The condition of a line device such that an application can perform any valid function on the line device, for example, wait for a call or make a call. By default, when a line device is first opened, it is in the blocked condition. The application receives a GCEV_UNBLOCKED event to indicate that the line device has moved to an unblocked condition from a previously blocked condition. See also *blocked*.

unsupervised transfer: A transfer in which the call is transferred without any consultation or announcement by the person transferring the call.

UUI: User-to-User Information. Proprietary messages sent to remote system during call establishment.

voice channel: Designates a bi-directional transfer of data for a single call between a voice device processing that call and the SCbus. Digitized voice from the T1/E1 interface device is transmitted over the SCbus to the voice receive (listen) channel for processing by the voice device. The voice device sends the response to the call over the voice transmit channel to an SCbus time slot that transmits this response to the T1/E1 interface device.

voice handle: SRL device handle associated with a voice channel; equivalent to the device handle returned from the voice library's **dx_open( )** function.

voice resource: See *voice channel*.

Index

P

R

S

T

U

W