Dialogic® Global Call API

Programming Guide

December 2007

05-2409-004

Copyright © 1996-2007, Dialogic Corporation. All rights reserved. You may not reproduce this document in whole or in part without permission in writing from Dialogic Corporation.

All contents of this document are furnished for informational use only and are subject to change without notice and do not represent a commitment on the part of Dialogic Corporation or its subsidiaries (“Dialogic”). Reasonable effort is made to ensure the accuracy of the information contained in the document. However, Dialogic does not warrant the accuracy of this information and cannot accept responsibility for errors, inaccuracies or omissions that may be contained in this document.

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

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

It is possible that the use or implementation of any one of the concepts, applications, or ideas described in this document, in marketing collateral produced by or on web pages maintained by Dialogic may infringe one or more patents or other intellectual property rights owned by third parties. Dialogic does not provide any intellectual property licenses with the sale of Dialogic products other than a license to use such product in accordance with intellectual property owned or validly licensed by Dialogic and no such licenses are provided except pursuant to a signed agreement with Dialogic. More detailed information about such intellectual property is available from Dialogic’s legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. **Dialogic encourages all users of its products to procure all necessary intellectual property licenses required to implement any concepts or applications and does not condone or encourage any intellectual property infringement and disclaims any responsibility related thereto. These intellectual property licenses may differ from country to country and it is the responsibility of those who develop the concepts or applications to be aware of and comply with different national license requirements.**

Dialogic, Diva, Eicon, Eicon Networks, Dialogic Pro, EiconCard and SIPcontrol, among others, are either registered trademarks or trademarks of Dialogic. Dialogic's trademarks may be used publicly only with permission from Dialogic. Such permission may only be granted by Dialogic’s legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. Any authorized use of Dialogic's trademarks will be subject to full respect of the trademark guidelines published by Dialogic from time to time and any use of Dialogic’s trademarks requires proper acknowledgement. Windows is a registered trademark of Microsoft Corporation in the United States and/or other countries. Other names of actual companies and products mentioned herein are the trademarks of their respective owners.

Publication Date: December 2007

Document Number: 05-2409-004

Contents

Revision History 9

About This Publication 11

Purpose 11
Applicability 11
Intended Audience 12
How to Use This Publication 12
Related Information 13

1 Product Description 15

1.1 Dialogic® Global Call API Software Overview 15
1.2 Dialogic® Global Call API Feature Categories 16
1.2.1 Call Control Features 16
1.2.2 Operation, Administration and Maintenance Features 17
1.3 Dialogic® Global Call API Architecture 17
1.3.1 Overview 18
1.3.2 Dialogic® Global Call API 19
1.4 Call Control Libraries 20
1.4.1 Starting Call Control Libraries 20
1.4.2 Call Control Library States 20
1.5 Dialogic® Global Call API Object Identifiers 21
1.5.1 Line Device Identifier 22
1.5.2 Call Reference Number 22
1.5.3 Object Identifiers and Resource Sharing Across Processes 23
1.5.4 Target Objects 23

2 Programming Models 27

2.1 Programming Models Overview 27
2.2 Asynchronous Mode Programming for Linux 27
2.3 Asynchronous Mode Programming for Windows® 28
2.3.1 Asynchronous Model Overview 28
2.3.2 Asynchronous Model with Event Handlers 29
2.3.3 Asynchronous with Windows® Callback Model 29
2.3.4 Asynchronous with Win32 Synchronization Model 30
2.3.5 Extended Asynchronous Programming Model 30

3 Call State Models 33

3.1 Call State Model Overview 33
3.2 Basic Call Model 33
3.2.1 Basic Call States at the Inbound Interface 34
3.2.2 Basic Call States at the Outbound Interface 35
3.2.3 Basic Call States for Call Termination 35
3.3 Basic Call Model Configuration Options 36
3.3.1 Call State Configuration 36
3.3.2 Call State Event Configuration 37
3.3.3 Call Acknowledgement Configuration 38

3.3.4 Call Proceeding Configuration 38
3.4 Basic Call Control in Asynchronous Mode 39
3.4.1 Inbound Calls in Asynchronous Mode 39
3.4.2 Outbound Calls in Asynchronous Mode 52
3.4.3 Call Termination in Asynchronous Mode 60
3.4.4 Handling Unsolicited Events 64
3.5 Advanced Call Control with Call Hold and Transfer 64
3.5.1 Advanced Call State Model Overview 64
3.5.2 Advanced Call States for Hold and Transfer 65
3.5.3 Call Hold 65
3.5.4 Call Transfer 66

4 Event Handling 71

4.1 Overview of Event Handling 71
4.2 Event Categories 71
4.3 Blocked and Unblocked Event Handling 72
4.4 Event Retrieval 73
4.5 Events Indicating Errors 74
4.6 Masking Events 74
4.7 Event Handlers 74
4.7.1 Event Handlers for Linux 74
4.7.2 Event Handlers for Windows® 75

5 Application Development Guidelines 77

5.1 General Programming Tips 77
5.2 Tips for SRL-Related Programming in a Linux Environment 78
5.3 Tips for Programming Drop and Insert Applications 78
5.4 Using the Dialogic® Global Call API with Dialogic® Digital Network Interface Boards 80
5.4.1 Routing Overview 80
5.4.2 Working with Flexible Routing Configurations 80
5.4.3 Handling Multiple Call Objects Per Channel in a Glare Condition 84

6 Error Handling 85

6.1 Error Handling Overview 85

7 Call Control 87

7.1 Call Progress Analysis when Using IP Technology 87
7.2 Call Progress Analysis when Using Dialogic® Digital Network Interface Boards 87
7.2.1 Call Progress Analysis Definition 87
7.2.2 Configuring Default Call Progress Analysis Parameters 88
7.2.3 Configuring Call Progress Analysis on a Per Call Basis 88
7.2.4 Setting Call Analysis Attributes on a Per Call Basis 90
7.2.5 Configuring Call Progress Analysis on a Per Channel Basis 91
7.2.6 Setting Call Analysis Attributes on a Per Channel Basis 92
7.2.7 Customizing Call Progress Tones on a Per Board Basis 92
7.3 Resource Routing 93
7.4 Feature Transparency and Extension 93
7.4.1 Feature Transparency and Extension Overview 93
7.4.2 Technology-Specific Feature Access 94
7.4.3 Technology-Specific User Information 95

8 Alarm Handling . . . 97
8.1 Alarm Handling Overview . . . 97
8.1.1 Alarm Management System Components . . . 97
8.2 Operation and Configuration of GCAMS . . . 99
8.2.1 Generation of Events for Blocking Alarms . . . 99
8.2.2 Generation of Alarm Events . . . 100
8.2.3 Configuration of Alarm Properties and Characteristics . . . 101
8.2.4 Starting and Stopping Alarm Transmission . . . 104
8.2.5 Retrieving Alarm Data . . . 104
8.3 Sample Alarm Scenarios . . . 105
8.3.1 Scenario 1: Application Notified of First and Last Blocking Alarm . . . 105
8.3.2 Scenario 2: Default Behavior for Alarm Notification . . . 107
8.3.3 Scenario 3: Alarm Transmission . . . 108

9 Real Time Configuration Management . . . 109
9.1 Real Time Configuration Manager Overview . . . 109
9.2 RTCM Components . . . 110
9.2.1 Customer Application Using Dialogic® Global Call API RTCM . . . 111
9.2.2 Dialogic® Global Call API RTCM . . . 111
9.2.3 RTCM Parameters . . . 112
9.3 Using RTCM Parameters . . . 112
9.3.1 Parameter Dependencies . . . 113
9.3.2 Parameter Definitions . . . 113
9.4 Getting and Setting Parameter Information . . . 114
9.4.1 GC_PARM_BLK Data Structure . . . 114
9.4.2 Control Parameters . . . 115
9.5 Handling RTCM Errors . . . 117
9.6 Configuration Procedure . . . 118
9.7 Sample Scenarios Using the RTCM API Functions . . . 119
9.7.1 Getting or Setting GCLib Configuration in Synchronous Mode . . . 120
9.7.2 Getting or Setting CCLib Configuration in Synchronous Mode . . . 121
9.7.3 Getting or Setting Line Device Configuration in Synchronous Mode . . . 122
9.7.4 Setting Line Device Configuration in Asynchronous Mode . . . 124
9.7.5 Setting Board Device Configuration in Asynchronous Mode (IP Technology) . . 125

10 Handling Service Requests . . . 127
10.1 Service Request Overview . . . 127
10.2 Service Request Components . . . 128
10.3 Service Request Data . . . 129
10.4 General Service Request Scenario . . . 130

11 Using Dialogic® Global Call API to Implement Call Transfer . . . 131
11.1 Introduction to Call Transfer . . . 131
11.1.1 Blind Call Transfer . . . 131
11.1.2 Supervised Call Transfer . . . 132
11.2 Call Transfer State Machine . . . 132

12 Building Applications . . . 139
12.1 Compiling and Linking in Linux . . . 139
12.1.1 Include Files . . . 139

12.1.2 Required Libraries 139
12.1.3 Variables for Compiling and Linking Commands 140
12.2 Compiling and Linking in Windows® 140
12.2.1 Include Files 140
12.2.2 Required Libraries 140
12.2.3 Variables for Compiling and Linking Commands 141
12.2.4 Dynamically Loaded Libraries 141

13 Debugging 143

Glossary 145

Index 153

Figures

1 Dialogic® Global Call API Architecture for IP Technology 18
2 Dialogic® Global Call API Architecture for E1/T1 and ISDN Technologies 19
3 Call Control Library States 21
4 Basic Asynchronous Inbound Call State Diagram 40
5 Basic Asynchronous Inbound Call Scenario 47
6 Incoming Call Scenario with Call Proceeding 48
7 Call Acknowledgement and Call Proceeding Done at the Application Layer 49
8 Call Proceeding Done by the Application Layer with Minimum Information Configured 50
9 Call Acknowledgement and Call Proceeding Done at Technology Call Control Layer 51
10 Call Acknowledgement Done by the Technology Call Control Layer and Call Proceeding Done by the Application 52
11 Basic Asynchronous Outbound Call State Diagram 54
12 Asynchronous Outbound Call Scenario 58
13 Asynchronous Outbound Call Scenario With Call Acknowledgement 59
14 Asynchronous Outbound Call Scenario With Overlap Sending 60
15 Asynchronous Call Tear-Down State Diagram 61
16 User Initiated Asynchronous Call Termination Scenario 63
17 Network Initiated Asynchronous Call Termination Scenario 63
18 Call State Transitions for Hold and Retrieve 66
19 Call State Model for Supervised and Unsupervised Transfers 68
20 Call Termination by the Network or Application During a Transfer 69
21 Architectural Diagram of Alarm Management Components 98
22 Notification of First and Last Blocking Alarm 106
23 Default Behavior for Alarm Notification 107
24 Alarm Transmission 108
25 Relationship of Customer Application, Dialogic® Global Call API RTCM, and RTCM Parameters 110
26 Run Time Configuration Procedure 119
27 Getting or Setting GCLib Configuration in Synchronous Mode 120
28 Getting or Setting CCLib Configuration in Synchronous Mode 121
29 Getting or Setting Line Device Configuration in Synchronous Mode 123
30 Setting Line Device Configuration in Asynchronous Mode (E1, T1 and ISDN Technology) . . 124
31 Setting Board Device Configuration in Asynchronous Mode (IP Technology) 125
32 Service Request Architecture 128
33 Generic Service Request Operation 130
34 Blind Call Transfer (Unsupervised Transfer) 132
35 Supervised Call Transfer 132
36 Call State Model for Blind Call Transfer at Party A 134
37 Call State Model for Blind Transfer at Party B 135
38 Call State Model for Supervised Transfer at Party A 136
39 Call State Model for Supervised Transfer at Party B 137
40 Call State Model for Supervised Transfer at Party C 138

Tables

1 Call Control Library States 21
2 Supported Target Types 23
3 Target Types and Target IDs 24
4 Target Object Availability 25
5 Obtaining Target IDs 26
6 Asynchronous Inbound Call State Transitions 41
7 Asynchronous Outbound Call State Transitions 55
8 Asynchronous Call Termination Call State Transitions 62
9 Unsolicited Events Requiring Signal Handlers 64
10 Handling Glare 84
11 Call Progress Analysis Settings and Possible Results 89
12 Comparison with Call Progress Analysis Using gc_SetParm( ) 92
13 Update Condition Flag and Dialogic® Global Call API Process 117
14 New Dialogic® Global Call API Transfer Call States 133

Revision History

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

Document No.	Publication Date	Description of Revisions
05-2409-004	December 2007	Made global changes to reflect Dialogic brand. Changed title to "Dialogic® Global Call API Programming Guide." Combined Windows® and Linux versions of this document so that this version supports both operating systems, and supersedes "Dialogic® Global Call API Programming Guide" (document number 05-2346-001). Building Applications chapter: Added note in Required Libraries section of Compiling and Linking in Linux about listing Dialogic® libraries before all other libraries.
05-2409-003	August 2006	Call Control Libraries section: Updated the library descriptions to identify the technologies/protocols that each library supports. Using Protocols (Flexible Routing) section: Removed incorrect reference to using the DM3 PDK Manager. Setting Call Analysis Attributes on a Per Call Basis section: Updated to indicate that PAMD_QUAL2TMP is not supported and to provide a pointer to a related Tech Note. Debugging chapter: Added reference to the "Runtime Trace Facility (RTF) Reference" chapter in the *Dialogic® Host Media Processing Diagnostics Guide*.
05-2409-002	October 2005	Global change: Added support for protocols that can be run on the E1 or T1 interfaces provided by Intel NetStructure® Digital Network Interface boards. Global change: Updates to recognize the Intel NetStructure® brand. Starting Call Control Libraries section: Added note about loading only the required call control libraries to keep the required memory footprint small. Overlap Sending section: Explicitly mentioned ISDN in the list of technologies that do not have messages to request more information. Configuring Default Call Progress Analysis Parameters section: Added a note that pre-connect call progress is enabled by default, regardless of the CPA setting in the CONFIG file Real Time Configuration Management chapter: Fixed several references to gc_util_insert_val( ) and gc_util_insert_ref( ) which should be gc_util_insert_parm_val( ) and gc_util_insert_parm_ref( ). Supervised Transfers section: Updated the call termination figure and added note to describe the unsolicited GCEV_CONNECTED event that is generated for a call when the new call being set up is terminated.
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
- Applicability
- Intended Audience
- How to Use This Publication
- Related Information

Purpose

This publication provides guidelines for using the Dialogic® 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 *Dialogic® Global Call API Library Reference* that provides details on the functions and parameters in the Dialogic® Global Call API library and the *Dialogic® Global Call Technology Guides* that provide IP-, E1/T1- and ISDN-specific information.

Dialogic® Host Media Processing (HMP) Software performs media processing tasks on general-purpose servers based on Dialogic® architecture without the need for specialized hardware. When installed on a system, Dialogic® HMP Software performs like a virtual Dialogic® DM3 board to the customer application, but media processing takes place on the host processor. In this document, the term "board" represents the virtual Dialogic® DM3 board, unless explictly noted otherwise. Dialogic® Digital Network Interface boards provide physical E1 and T1 interfaces for applications that require E1/T1 network connectivity.

Applicability

This document version (05-2409-004) is published for Dialogic® Host Media Processing Software Release 3.1LIN.

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

Intended Audience

This 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 Dialogic® Global Call API software.

This publication assumes that you are familiar with the both the Linux and Windows® operating systems and the C programming language.

The information in this guide is organized as follows:

- Chapter 1, "Product Description" provides an overview of the Dialogic® Global Call API development software.
- Chapter 2, "Programming Models" describes the supported programming models in the Linux and Windows® environments.
- Chapter 3, "Call State Models" describes the call state models used by Dialogic® Global Call API.
- Chapter 4, "Event Handling" describes how to handle Dialogic® Global Call API events.
- Chapter 5, "Application Development Guidelines" provides guidelines for developing applications that use Dialogic® Global Call API.
- Chapter 6, "Error Handling" describes the error handling facilities provided by Dialogic® Global Call API.
- Chapter 7, "Call Control" describes basic call control capabilities, resource routing and feature extensions provided by Dialogic® Global Call API.
- Chapter 8, "Alarm Handling" describes how Dialogic® Global Call API can be used to handle alarms.
- Chapter 9, "Real Time Configuration Management" describes how Dialogic® Global Call API 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 Dialogic® Global Call API.
- Chapter 11, "Using Dialogic® Global Call API 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 Dialogic® Global Call API 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

See the following for additional information:

- *http://www.dialogic.com/manuals/* (for Dialogic® product documentation)
- *http://www.dialogic.com/support/* (for Dialogic technical support)
- *http://www.dialogic.com/* (for Dialogic® product information)

Product Description 1

This chapter describes the Dialogic® Global Call API software. Topics include:

- Dialogic® Global Call API Software Overview 15
- Dialogic® Global Call API Feature Categories 16
- Dialogic® Global Call API Architecture 17
- Call Control Libraries 20
- Dialogic® Global Call API Object Identifiers 21

1.1 Dialogic® Global Call API Software Overview

Dialogic® Global Call API 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 Dialogic® Global Call API software facilitates the exchange of call control messages between the telephone network and any network-enabled applications. Dialogic® Global Call API software enables developers to create applications that can work with signaling systems worldwide, regardless of the network to which the applications are connected. The Dialogic® Global Call API software is well suited 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, Dialogic® Global Call API software offers a consistent, high-level interface to the user and handles each country's unique protocol requirements transparently to the application.

The Dialogic® Global Call API software comprises three major components:

Dialogic® Global Call API 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.

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

Dialogic® Global Call API Protocols
: Network signaling protocols, such as T1 Robbed Bit, E1 CAS, ISDN, QSIG, IP H.323 and SIP can be invoked by the Dialogic® Global Call API to facilitate call control.

1.2 Dialogic® Global Call API Feature Categories

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

- Call Control Features
- Operation, Administration and Maintenance Features

1.2.1 Call Control Features

The Dialogic® Global Call API 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.

Advanced Call Model
: Defines the behavior for advanced features, such as hold and transfer. These capabilities are provided to support technologies and protocols that support such features, for example, Supervised Transfer. The implementation of these capabilities is based on a more advanced call state model. See Section 3.5, “Advanced Call Control with Call Hold and Transfer” for more information. The advanced call model applies only to E1/T1 and ISDN technologies, not IP technology, which uses a different scheme for features such as call transfer. See the *Dialogic® Global Call IP Technology Guide*.

Call Progress and Call Analysis
: Provides the capabilities for handling pre-connect (Call Progress) information that reports the status of the call connection, such as, busy, no dial tone or no ringback, and post connect (Call Analysis) information that reports the destination party’s media type, for example, voice, answering machine, or fax modem. This information is determined by the detection of tones defined specifically for this purpose. See Section 7.2, “Call Progress Analysis when Using Dialogic® Digital Network Interface Boards” for more information. The call progress and call analysis feature applies only to E1/T1 and ISDN technologies, not IP technology.

Feature Transparency and Extension (FTE)
: Provides the ability to extend the capabilities of Dialogic® Global Call API to handle features that are specific to a particular technology so that those features are accessible via the Dialogic® Global Call API interface. For example, for ISDN applications, Dialogic® Global Call API supports supplementary services such as Overlap Send, Overlap Receive, Any Message, Any IE, and User-to-User messaging. See Section 7.4, “Feature Transparency and Extension” for more information.

1.2.2 Operation, Administration and Maintenance Features

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

Error Handling Functionality
: When an error occurs, Dialogic® Global Call API 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.

Dialogic® Global Call API Alarm Management System (GCAMS)
: Provides the ability to manage alarms. GCAMS provides Dialogic® Global Call API 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.

Dialogic® Global Call API 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 *Dialogic® Global Call API Library Reference* for more information about these functions.

Debugging Facilities
: Dialogic® Global Call API provides powerful debugging capabilities for troubleshooting protocol-related problems, including the ability to generate a detailed log file. See the appropriate Dialogic® Global Call API Technology Guide for information on the debugging facilities available when using Dialogic® Global Call API with each technology.

1.3 Dialogic® Global Call API Architecture

The Dialogic® Global Call API development software architecture is based on the Dialogic® architecture that supports Dialogic® Host Media Processing (HMP) Software and Dialogic® DM3 hardware. The architecture is described in the following topics:

- Overview
- Dialogic® Global Call API

1.3.1 Overview

Figure 1 shows a system-level view of the Dialogic® Global Call API architecture for IP technology and Figure 2 shows the Dialogic® Global Call API architecture for E1/T1 and ISDN technologies on Dialogic® DM3 hardware.

Figure 1. Dialogic® Global Call API Architecture for IP Technology

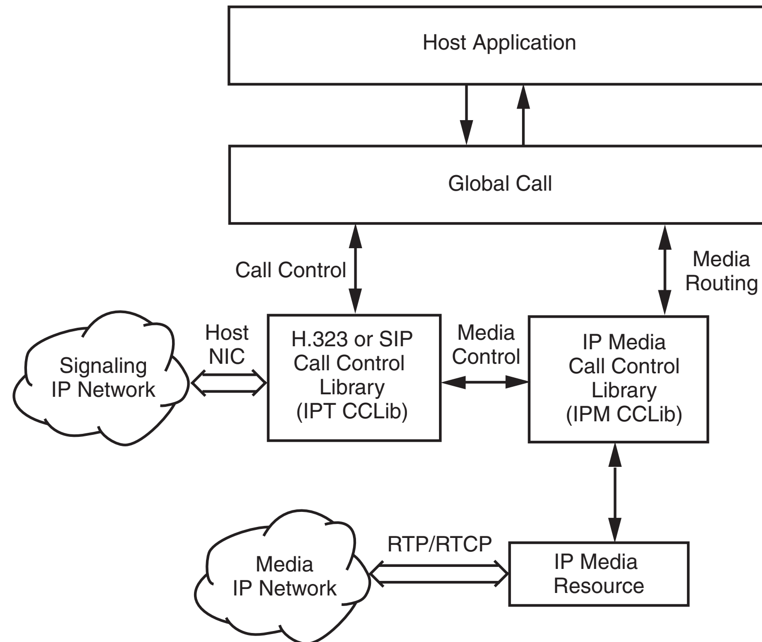
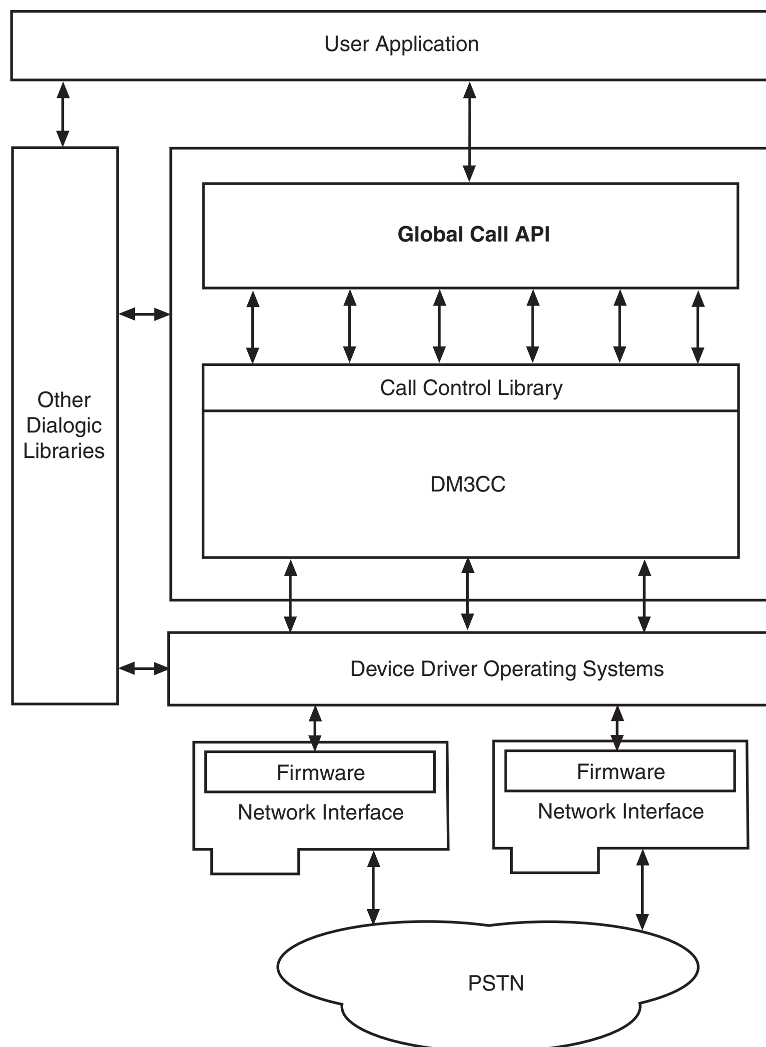


Figure 2. Dialogic® Global Call API Architecture for E1/T1 and ISDN Technologies



1.3.2 Dialogic® Global Call API

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

The Dialogic® 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 *Dialogic® 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 Dialogic® Global Call API library. The call control libraries currently supported by the Dialogic® Global Call API for Dialogic® HMP Software are as follows:

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

GC_DM3CC_LIB
: The call control library that controls access to network interfaces on Digital Network Interface boards. This library is used for call control using ISDN and CAS/R2MF (PDK protocols) signaling on Digital Network Interface boards.

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 and is used in conjunction with GC_IPM_LIB.

GC_IPM_LIB
: The call control library that provides access to IP media resources. This library is used for H323/SIP call control signaling and is used in conjunction with GC_H3R_LIB.

1.4.1 Starting Call Control Libraries

Call control libraries must be started before they can be used by the Dialogic® Global Call API 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 Dialogic® Global Call API. See the *Dialogic® Global Call API Library Reference* for more information about the **gc_Start( )** function.

Note: Invoking **gc_Start(NULL)** loads all call control libraries and consequently the memory footprint includes memory that is allocated for all call control libraries. To reduce the memory footprint, selective loading of call control libraries is recommended. For more information and an example, see the **gc_Start( )** function in the *Dialogic® Global Call API Library Reference*.

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 remains in the Configured state.

Figure 3. Call Control Library States

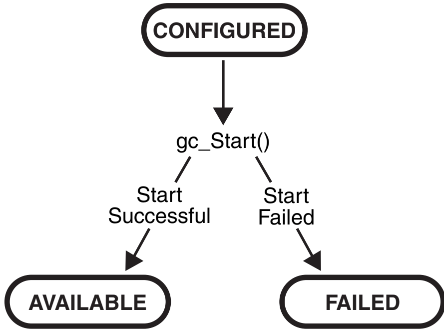


Table 1 describes the different states of a call control library.

Table 1. Call Control Library States

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

Each configured call control library is assigned an ID number by Dialogic® Global Call API. 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 *Dialogic® Global Call API Library Reference* for detailed information about these functions.

1.5 Dialogic® Global Call API Object Identifiers

The Dialogic® 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 Dialogic® Global Call API Service Request (GCSR) operate on a basic entity called a Dialogic® Global Call API 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 Dialogic® Global Call API 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 Dialogic® Global Call API, 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( )** 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 Dialogic® Global Call API 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 *Dialogic® Global Call API Library Reference*.

Outbound calls
: The CRN is assigned via the **gc_MakeCall( )** function. For more information on this function, see the *Dialogic® 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 Dialogic® 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. Unpredictable results may occur if this advice is not followed.

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 Dialogic® Global Call API 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 Dialogic® Global Call API 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 appropriate Dialogic® Global Call API Technology Guide for more information.

The possible software modules include:

- GCLib
- CCLib

- Protocol
- Firmware

The possible entities include:

System
: NIC for IP technology; all physical boards for E1, T1 and ISDN technologies

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 Dialogic® Global Call API 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 Dialogic® Global Call API 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)
- Dialogic® Global Call API library ID (GCGV_LIB)
- call control library ID (CCLib ID)
- protocol ID

The types and IDs for target objects are defined at the Dialogic® Global Call API 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	Dialogic® Global Call API library module target object.
GCTGT_CCLIB_SYSTEM † ‡	CCLib ID	Call control library module target object.
GCTGT_GCLIB_NETIF	Dialogic® Global Call API Line device ID	Network interface target object in Dialogic® Global Call API Library module.
GCTGT_CCLIB_NETIF	Dialogic® Global Call API Line device ID	Network interface target object in call control library module.
† For E1, T1 and ISDN technologies only. ‡ 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_GetConfigData( )** (E1, T1 and ISDN technologies only), **gc_SetConfigData( )**, **gc_ReqService( )**, and **gc_RespService( )**.		

Table 3. Target Types and Target IDs (Continued)

Target Type	Target ID	Description
GCTGT_GCLIB_CHAN	Dialogic® Global Call API Line device ID	Channel target object in Dialogic® Global Call API library module.
GCTGT_CCLIB_CHAN	Dialogic® Global Call API Line device ID	Channel target object in call control library module.
GCTGT_GCLIB_CRN	Dialogic® Global Call API CRN	CRN target object in Dialogic® Global Call API library module.
GCTGT_CCLIB_CRN	Dialogic® Global Call API CRN	CRN target object in call control library module.

† For E1, T1 and ISDN technologies only.
‡ 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_GetConfigData( )** (E1, T1 and ISDN technologies only), **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 Dialogic® 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 GCTGT_CCLIB_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( )**

† For E1, T1 and ISDN technologies only.

Retrieving Target IDs

Before the Dialogic® Global Call API 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.
Dialogic® Global Call API Line Device ID †	After a line device is opened, the CCLib ID and protocol ID (if applicable) associated with this line device can be obtained by the **gc_GetConfigData( )** function with the set ID and parameter ID as (GCSET_CCLIB_INFO, GCPARM_CCLIB_ID) and (GCSET_PROTOCOL, GCPARM_PROTOCOL_ID).
Dialogic® Global Call API CRN	After a call target object is created, its target object ID (that is, the Dialogic® Global Call API CRN) will be an output of the **gc_MakeCall( )** function or provided by the metaevent associated with the GCEV_OFFERED event.
† For E1, T1 and ISDN technologies only.	

Programming Models 2

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

- Programming Models Overview 27
- Asynchronous Mode Programming for Linux. 27
- Asynchronous Mode Programming for Windows® 28

2.1 Programming Models Overview

The Dialogic® Global Call API 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 *Dialogic® Standard Runtime Library API Programming Guide*.

2.2 Asynchronous Mode Programming for Linux

Asynchronous mode programming is characterized by allowing other processing to take place while a function executes. In asynchronous mode programming, multiple channels are handled in a single process rather than in separate processes as required in synchronous mode programming.

An asynchronous mode function typically receives an event from the Dialogic® Standard Runtime Library (SRL) indicating completion (termination) of the function in order for the application to continue processing a call on a particular channel. A function called in the asynchronous mode returns control to the application after the request is passed to the device driver. A termination event is returned when the requested operation completes.

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( )**, the **gc_ResetLineDev( )** function should be called. If this is not done, there are potential race conditions under which the application may crash with a segmentation fault.

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

Asynchronous (Polled)

In this model, the application polls for or waits for events using the **sr_waitevt( )** function. 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. Typically, the polled model is used for applications that do not need to use event handlers to process events.

Asynchronous with Event Handlers

The Asynchronous with Event Handlers model may be run in non-signal mode only. Event handlers can be enabled or disabled for specific events on specific devices; see Chapter 4, "Event Handling" for details.

2.3 Asynchronous Mode Programming for Windows®

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

2.3.1 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 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 model
- 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 77 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 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.

2.3.2 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 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 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.

2.3.3 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 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

Dialogic® Global Call API events and optionally for non-Dialogic® Global Call API 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 *Dialogic® Standard Runtime Library API Programming Guide*.

2.3.4 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 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 *Dialogic® Standard Runtime Library API Programming Guide*.

2.3.5 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 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 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 the Dialogic® Global Call API. Topics include the following:

- Call State Model Overview 33
- Basic Call Model 33
- Basic Call Model Configuration Options 36
- Basic Call Control in Asynchronous Mode 39
- Advanced Call Control with Call Hold and Transfer 64

3.1 Call State Model Overview

The Dialogic® Global Call API 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 the Dialogic® Global Call API 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.

Get More Information (GCST_GETMOREINFO)
: This state exists for an incoming call when the network has received an acknowledgement of the call establishment request, which permits the network to send additional call information (if any) in the overlap mode. The application is waiting for more information, typically called party number digits. (This state is optional and may not be supported in all technologies. See the appropriate Dialogic® Global Call API Technology Guide for information.) This state applies to E1, T1 and ISDN technologies only.

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; for example, in the case of ISDN, a CALL_PROCEEDING message is sent. The application can now accept or answer the call. (This state is optional and may not be supported in all technologies. See the appropriate Dialogic® Global Call API Technology Guide for information.)

Call Accepted (GCST_ACCEPTED)
: 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 request more information,acknowledge the call, accept the call or answer the call.

Send More Information (GCST_SENDMOREINFO)
: This state exists for an outgoing call when the user has received an acknowledgement of the call establishment request that permits or requests the user to send additional call information to the network in overlap mode. The information, typically digits, is in the process of being prepared for transfer or being transferred across the telephony network (overlap sending or partial dialing). (This state is optional and may not be supported in all technologies. See the appropriate Dialogic® Global Call API Technology Guide for information.) This state applies to E1, T1 and ISDN technologies only.

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 appropriate Dialogic® Global Call API 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_CALLROUTING_STATE (for E1, T1, and ISDN technologies only)
- GCMSK_DETECTED_STATE
- GCMSK_GETMOREINFO_STATE (for E1, T1, and ISDN technologies only)
- GCMSK_PROCEEDING_STATE
- GCMSK_SENDMOREINFO_STATE (for E1, T1, and ISDN technologies only)

See the *Dialogic® 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
- GCMSK_REQMOREINFO (for E1, T1, and ISDN technologies only)

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

3.3.3 Call Acknowledgement Configuration

Note: This functionality applies to E1, T1 and ISDN technologies only.

When an incoming call is received, an acknowledgement is typically sent to the remote side to indicate that the call was received. In some technologies, if the incoming call does not have sufficient information, this acknowledgement also indicates to the remote side that more information is required to proceed with the call (see Section 3.4.1.8, "Overlap Receiving" for more information). Either the technology call control layer or the application can be configured to send the acknowledgement. This configuration can be set by the application issuing the **gc_SetConfigData( )** function. The set ID used in this context is GCSET_CALL_CONFIG and the relevant parm ID is:

GCPARM_CALLACK
: Specify whether call acknowledgement is provided by the application or the technology call control layer.

The GCPARM_CALLACK 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 *Dialogic® Global Call API Library Reference* for more information on the **gc_SetConfigData( )** function.

3.3.4 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 *Dialogic® 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

Note: For E1, T1 and ISDN technologies, the Advanced Call Model includes call states associated with holding, retrieving and transferring calls. See Section 3.5, "Advanced Call Control with Call Hold and Transfer" for more information.

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:

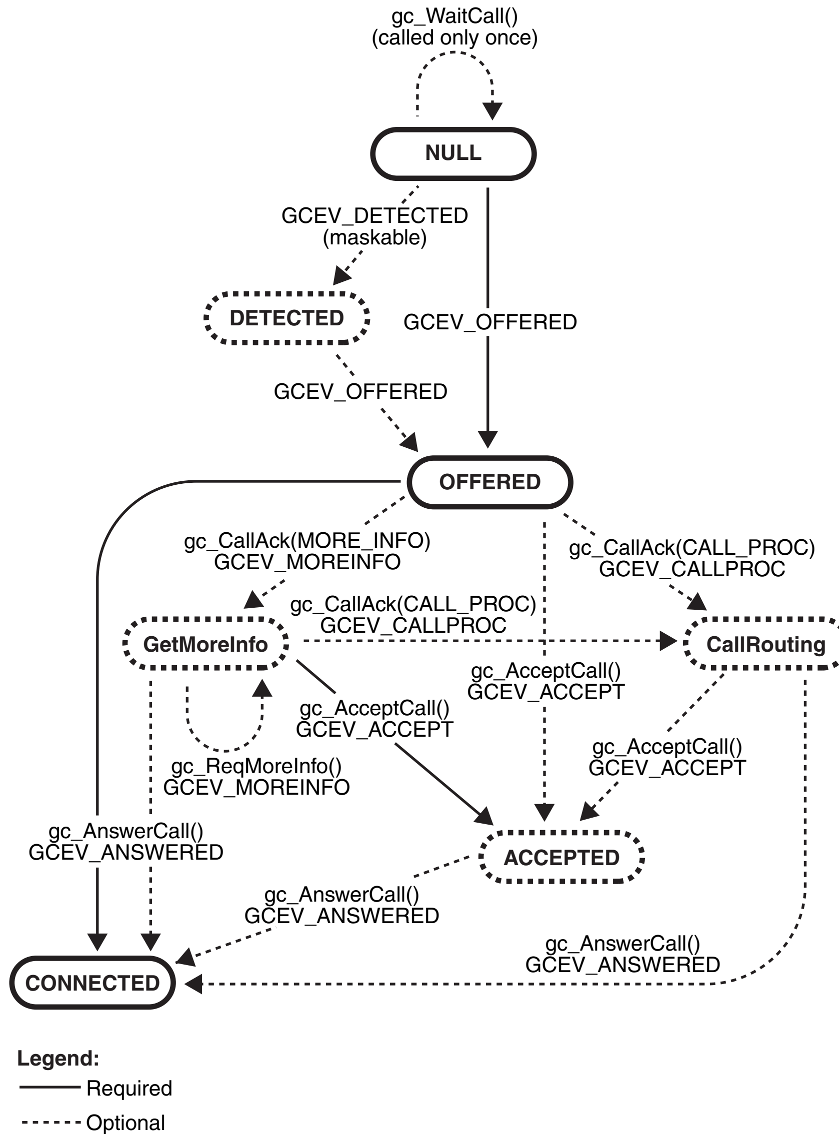
- Inbound Calls in Asynchronous Mode Overview
- Channel Initialization
- Call Detection
- Call Offered
- Call Routing
- Call Acceptance
- Call Establishment
- Overlap Receiving (for E1, T1, and ISDN technologies only)
- Call Failure
- Abandoned Calls
- Inbound Call Scenarios in Asynchronous Mode

3.4.1.1 Inbound Calls in Asynchronous Mode Overview

Figure 4 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 4.

See Table 6, "Asynchronous Inbound Call State Transitions", on page 41 for a summary of the call state transitions.

Figure 4. Basic Asynchronous Inbound Call State Diagram



Note: In Figure 4, the GetMoreInfo state and all transitions to/from that state apply to E1, T1, and ISDN technologies only.

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, GetMoreInfo †, 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, GetMoreInfo † **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, GetMoreInfo †,CallRouting, Dialing, SendMoreInfo †,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,
GetMoreInfo (GCST_GETMOREINFO) † Maskable	**Previous**: Offered **Next**: GCEV_ANSWERED -> Connected state GCEV_MOREINFO -> GetMoreInfo state GCEV_ACCEPT -> Accepted state GCEV_CALLPROC -> CallRouting state GCEV_DISCONNECTED -> Disconnected state GCEV_DROPCALL -> Idle state	**gc_ReqMoreInfo( )**, **gc_CallAck( )**, **gc_AnswerCall( )**, **gc_AcceptCall( )**, **gc_DropCall( )**	GCEV_DISCONNECTED, GCEV_DROPCALL, GCEV_ACCEPT, GCEV_ANSWERED, GCEV_MOREINFO or GCEV_CALLPROC
† Applies to E1, T1 and ISDN technology only.			

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_MOREINFO -> GetMoreInfo 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, GCEV_MOREINFO †, GCEV_CALLPROC
† Applies to E1, T1 and ISDN technology only.			

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, as well as to channel states (service status in T1 ISDN, bit states in CAS). 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.

When using E1, T1 and ISDN technology, if the incoming call does not have sufficient information, the call is offered to the application when all the required information is received. If the technology is configured to accept minimum information, the call is offered to the application when the specified minimum amount of information is received. In this case, the application must request additional information if required. See Section 3.4.1.8, “Overlap Receiving” for more information.

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 Overlap Receiving

Note: This functionality applies to E1, T1 and ISDN technologies only.

After an incoming call has been received, the call is offered to the application based on the call acknowledgement configuration and the availability of information required for proceeding with the call. If the incoming call is in en-bloc mode where all the information required for processing the call is present, the call is offered to the application. Otherwise, the call is offered to the application based on the following configurations:

Call Acknowledgement
: If the application is configured to send the call acknowledgement, the call is immediately offered to the application regardless of the amount of information available. The application can then request and collect more information as required. If the technology call control layer is configured to send the call acknowledgement, then the call is offered to the application based on the minimum amount of information specified.

Minimum Information Specified
: If the incoming call does not have sufficient information, the call is offered to the application based on the amount of information required. If the technology is configured to accept minimum information, the call is offered to the application only after the specified minimum amount of information is received. Thereafter, the application can request and collect more information as required. If the technology is not configured to accept minimum information,

then the call is offered to the application regardless of the amount of information available. The application can then request and collect more information as required.

The following sections describe various configurations operating in overlap receiving mode.

Scenario 1

In this scenario, the application is configured to acknowledge the incoming call and send a call proceeding indication after sufficient information has been received. When an incoming call is detected, the call is immediately offered to the application regardless of the amount of information available to proceed with the call.

When the call is in the Offered state (after the generation of the unsolicited GCEV_OFFERED event), the application sends an acknowledgement for the incoming call by issuing a **gc_CallAck(GCACK_SERVICE_INFO)** function. The application may selectively retrieve call information, such as Destination address and Origination address (caller ID) by issuing the **gc_GetCallInfo( )** function. If more information is still required, the **gc_ReqMoreInfo( )** function is issued to request more information. When the information is received, the GCEV_MOREINFO event is generated again. When all the required information is received, the application may send a call proceeding indication to the remote side by issuing the **gc_CallAck( )** function. Otherwise, the application can choose to accept or answer the call.

Scenario 2

In this scenario, the technology call control layer is configured to acknowledge the incoming call and send a call proceeding indication after sufficient information has been received. When an incoming call is detected, the technology call control layer immediately sends an acknowledgement. If the minimum amount of information required is specified, then the call is offered to the application only after the minimum amount of information required is received. After the call is offered to the application, the address information can be retrieved to determine if more information is required. If more information is required, a **gc_CallAck(GCACK_SERVICE_INFO)** function must be issued. Since an acknowledgement was already sent out, nothing is sent to the remote side at this time. However, if the minimum amount of information is not specified, then the technology control layer requests and collects more information. After all the maximum amount of information expected is received, the technology control layer sends a call proceeding indication to the remote side. The call is then offered to the application, which can then accept or answer the call.

Scenario 3

In this scenario, the technology call control layer is configured to acknowledge the incoming call and the application is configured to send a call proceeding indication after sufficient information has been received. When an incoming call is detected, the technology call control layer immediately sends an acknowledgement. If the minimum amount of information required is specified, then the call is offered to the application only after the minimum amount of information required is received. Otherwise the call is immediately offered to the application.

When the call is in the Offered state (after generation of the unsolicited GCEV_OFFERED event), the application may selectively retrieve call information, such as the Destination and Origination

address (caller ID) by issuing the **gc_GetCallInfo( )** function. If more information is required, the application may also request more address information using the **gc_CallAck(GCACK_SERVICE_INFO)** function. Since an acknowledgement was already sent out, no acknowledgement is sent to the remote side at this time. When the additional information is received, the GCEV_MOREINFO event is generated. If more information is still required, the **gc_ReqMoreInfo( )** function is issued to request more information. When the additional information is received, the GCEV_MOREINFO event is generated again. When all the required information is received, the application may send a call proceeding indication to the remote side by issuing the **gc_CallAck(GCACK_SERVICE_PROC)** function. Otherwise, the application can choose to accept or answer the call.

Scenario 4

In this scenario, the application is configured to acknowledge the incoming call and the technology call control layer is configured to send a call proceeding indication after sufficient information has been received. When an incoming call is detected, the call is offered to the application regardless of the amount of information available.

When the call is in the Offered state (after generation of the unsolicited GCEV_OFFERED event), the application sends an acknowledgement for the incoming call by issuing a **gc_CallAck(GCACK_SERVICE_INFO)**. The application may selectively retrieve call information, such as Destination address and Origination address (caller ID) by issuing the **gc_GetCallInfo( )** function. If more information is still required, the **gc_ReqMoreInfo( )** function is issued to request more information. When the information is received, the GCEV_MOREINFO event is generated again. When all the required information is received, the technology call control layer sends a call proceeding indication to the remote side. The application may also attempt to send a call proceeding indication to the remote side in case the technology call control layer hasn't done so. The application can then choose to accept or answer the call.

3.4.1.9 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 *Dialogic® Global Call API Library Reference*).

Forced Release (applies to E1, T1 and ISDN technologies only)

From the Accepted state, not all protocols support a forced release of the line, that is, issuing a **gc_DropCall( )** function after a **gc_AcceptCall( )** function. If a forced release is not supported and is attempted, the function will fail and an error will be returned. To recover, the application should issue the **gc_AnswerCall( )** function followed by **gc_DropCall( )** and **gc_ReleaseCallEx( )** functions. However, any time a GCEV_DISCONNECTED event is received in the Accepted state, the **gc_DropCall( )** function can be issued.

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 *Dialogic® Global Call API Library Reference*.

3.4.1.10 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.

3.4.1.11 Inbound Call Scenarios in Asynchronous Mode

This section shows various asynchronous inbound call scenarios. For call scenarios used by a specific signaling protocol, check the Dialogic® Global Call API Technology Guide for that technology.

Figure 5 shows a basic asynchronous call scenario for an incoming call.

Figure 5. Basic Asynchronous Inbound Call Scenario

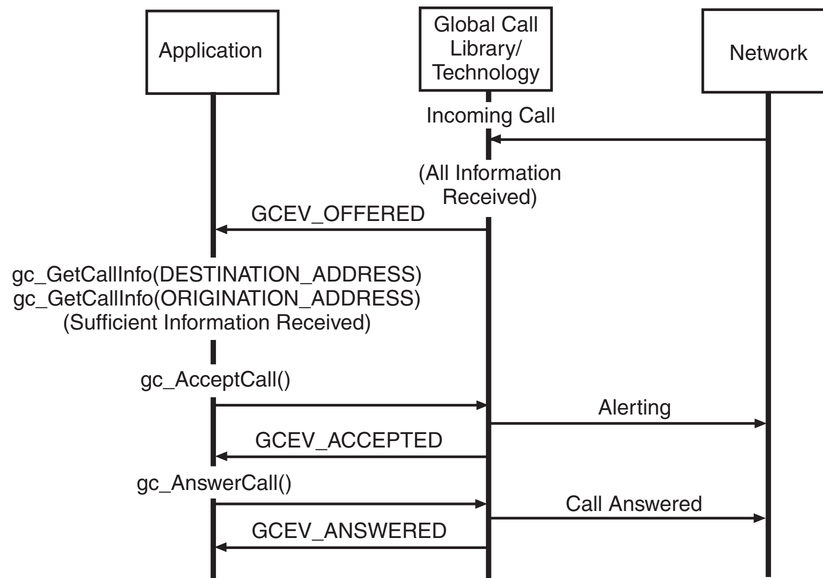


Figure 6 shows an asynchronous call scenario for an incoming call with call proceeding.

Figure 6. Incoming Call Scenario with Call Proceeding

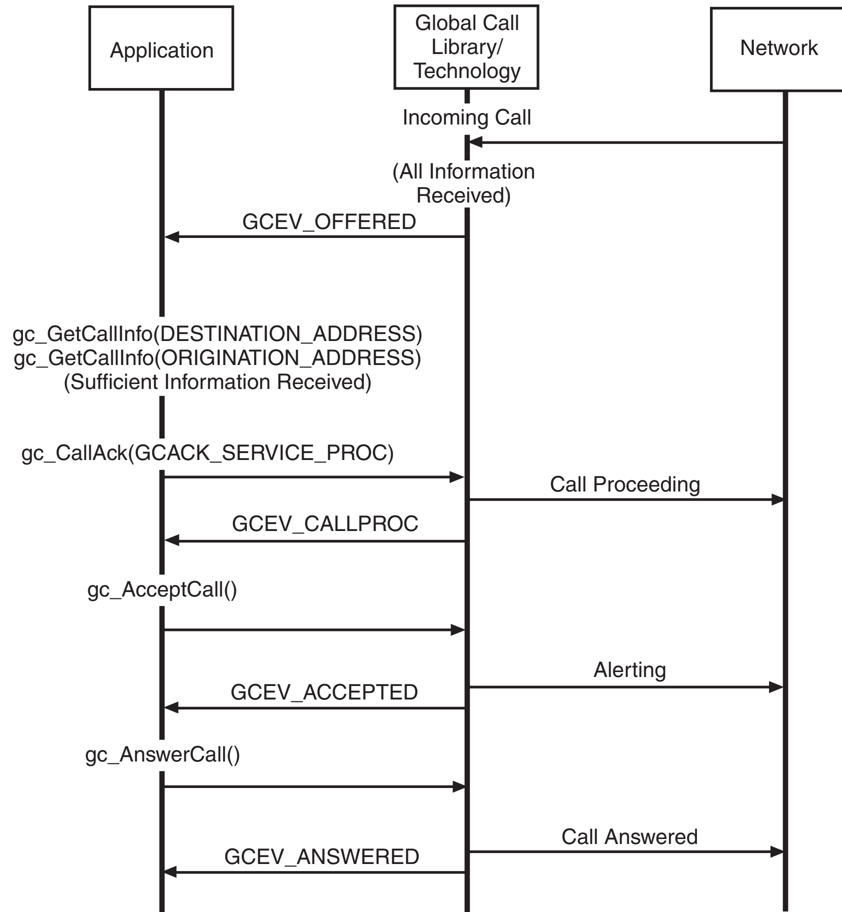


Figure 7 shows an asynchronous call scenario for an incoming call with call acknowledgement and call proceeding controlled by the application. This scenario applies to E1, T1 and ISDN technologies only.

Figure 7. Call Acknowledgement and Call Proceeding Done at the Application Layer

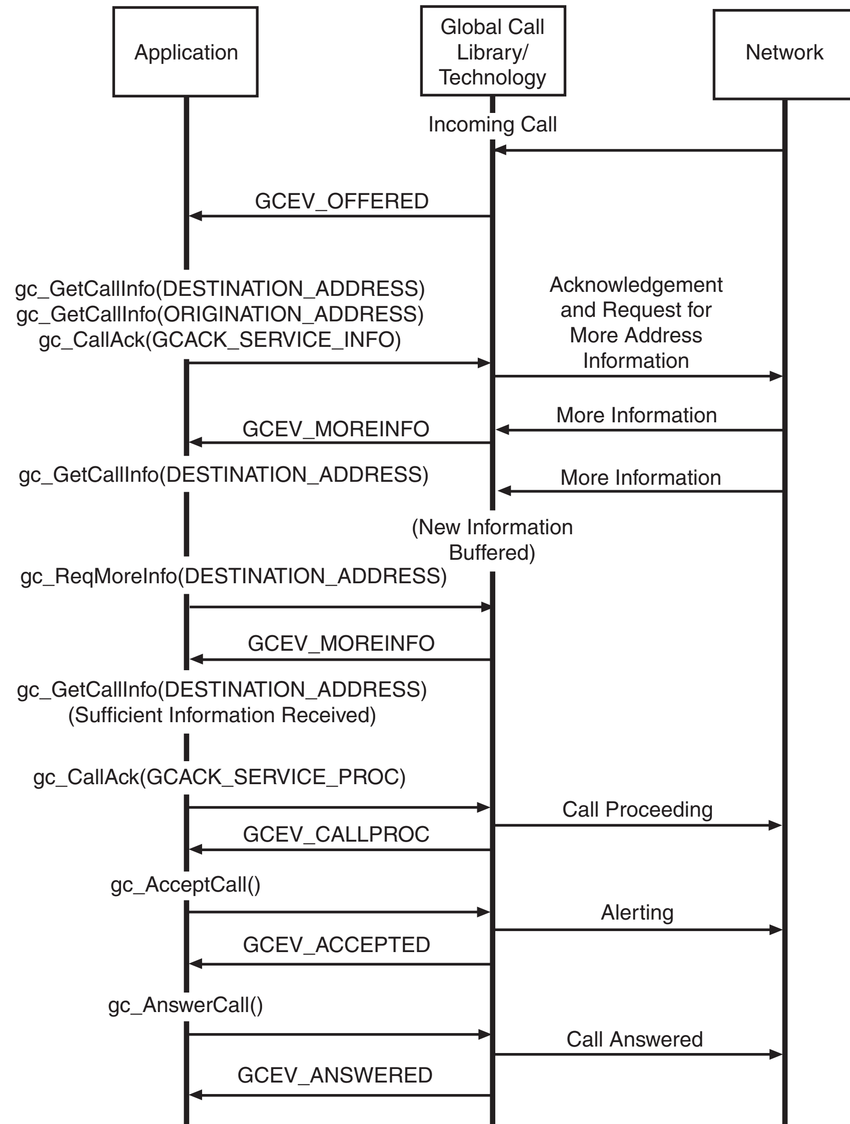


Figure 8 shows an asynchronous call scenario for an incoming call with call proceeding controlled by the application with the minimum information configuration. This scenario applies to E1, T1 and ISDN technologies only.

Figure 8. Call Proceeding Done by the Application Layer with Minimum Information Configured

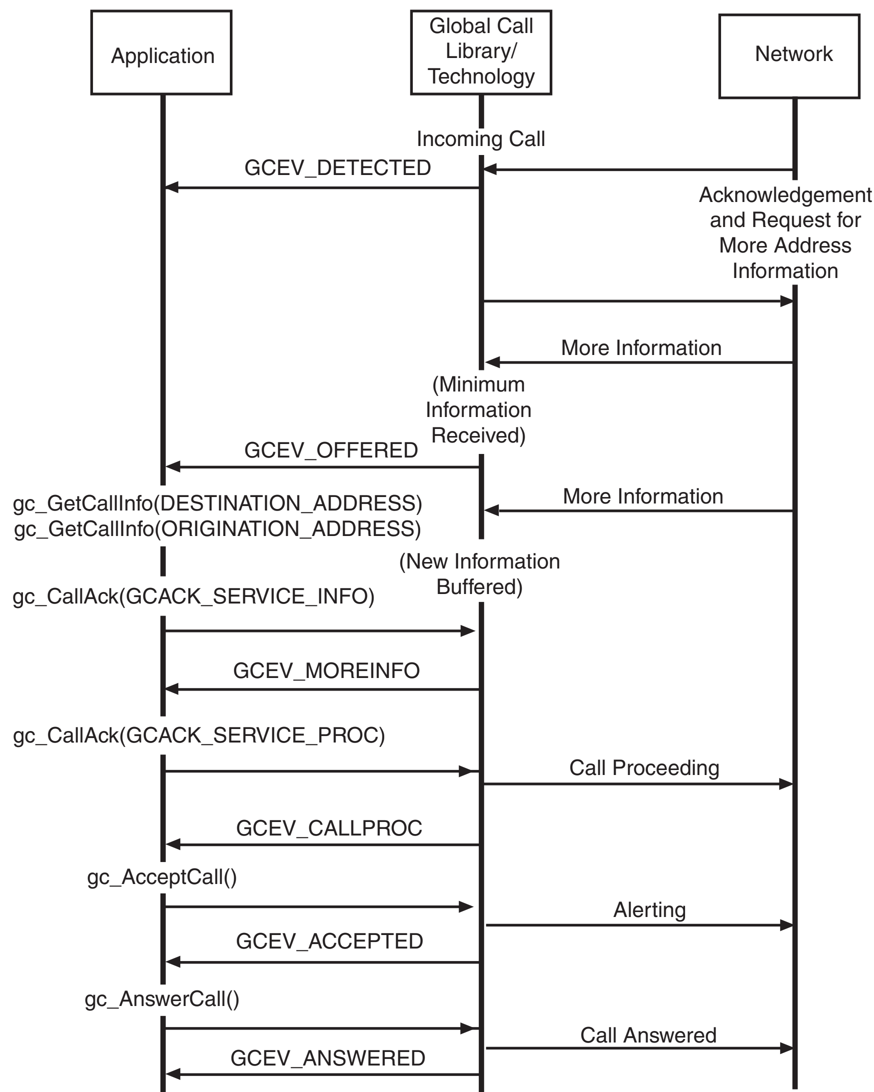
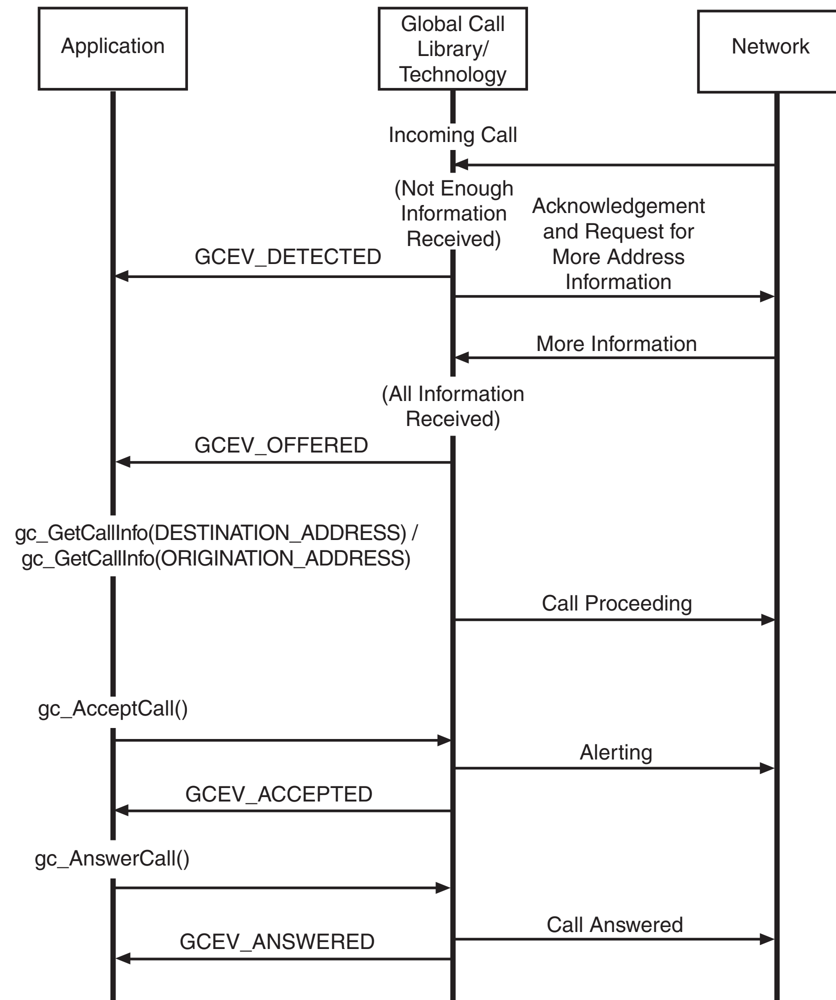


Figure 9 shows an asynchronous call scenario for an incoming call with call acknowledgement and call proceeding controlled by the call control layer.

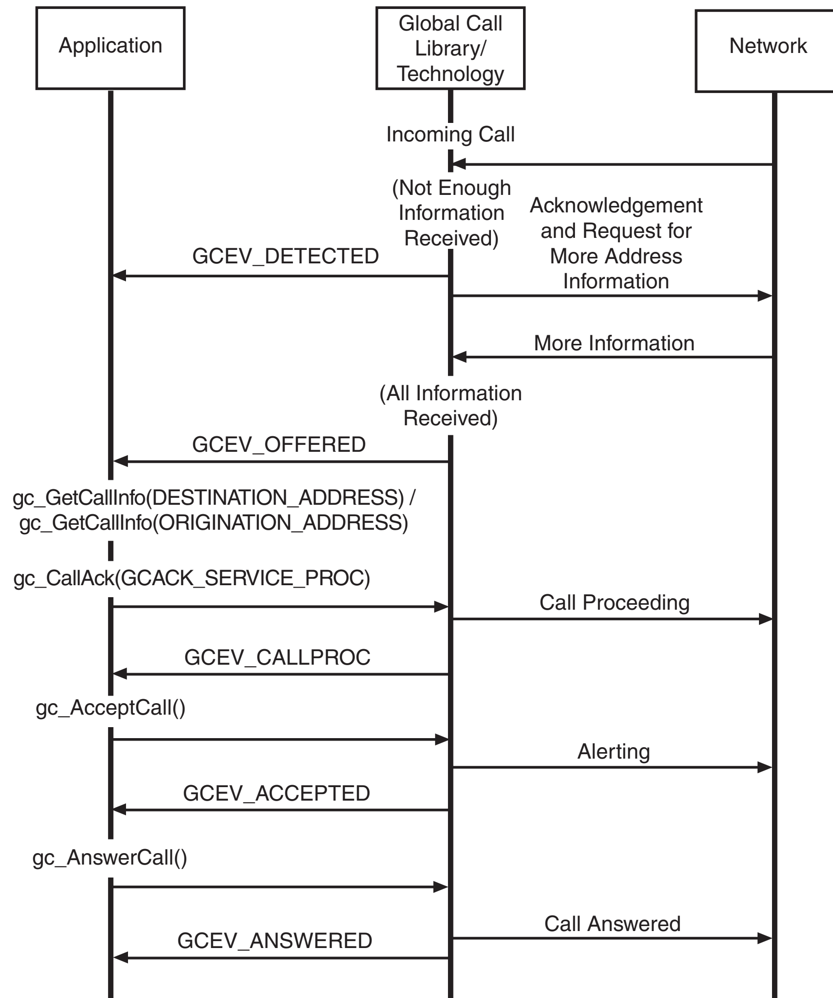
Figure 9. Call Acknowledgement and Call Proceeding Done at Technology Call Control Layer



Note: In Figure 9, the “Acknowledgement and Request for More Address Information” applies to E1, T1, and ISDN technologies only.

Figure 10 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 10. Call Acknowledgement Done by the Technology Call Control Layer and Call Proceeding Done by the Application



Note: In Figure 10, the “Acknowledgement and Request for More Address Information” applies to E1, T1, and ISDN technologies only.

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:

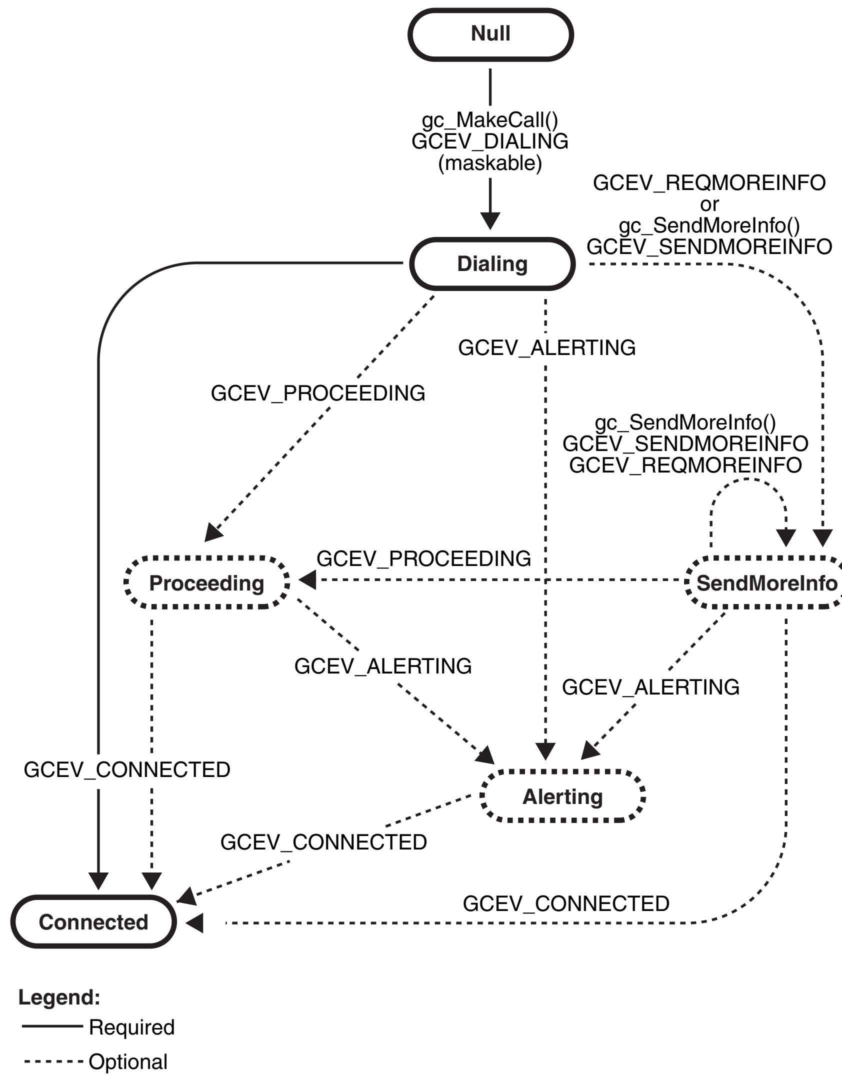
- Outbound Calls in Asynchronous Mode Overview
- Channel Initialization

- Call Dialing
- Call Proceeding
- Call Alerting
- Call Connected
- Overlap Sending (for E1, T1, and ISDN technologies only)
- Call Failure
- Outbound Call Scenarios in Asynchronous Mode

3.4.2.1 Outbound Calls in Asynchronous Mode Overview

Figure 11 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 11. Basic Asynchronous Outbound Call State Diagram



Note: In Figure 11, the SendMoreInfo state and all transitions to/from that state apply to E1, T1, and ISDN technologies only.

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, SendMoreInfo † **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_REQMOREINFO -> SendMoreInfo state † GCEV_SENDMOREINFO -> SendMoreInfo state † GCEV_DISCONNECTED -> Disconnected state GCEV_DROPCALL -> Idle state	**gc_SendMoreInfo( )** † **gc_DropCall( )**	GCEV_CONNECTED, GCEV_ALERTING, GCEV_REQMOREINFO †, 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, SendMoreInfo † **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
SendMoreInfo (GCST_SENDMOREINFO) † Maskable	**Previous**: Dialing **Next**: GCEV_CONNECTED -> Connected state GCEV_PROCEEDING -> Proceeding state. GCEV_DISCONNECTED -> Disconnected state GCEV_DROPCALL -> Idle state	**gc_SendMoreInfo( )** **gc_DropCall( )**	GCEV_DISCONNECTED, GCEV_DROPCALL, GCEV_PROCEEDING GCEV_CONNECTED
† Applies to E1, T1 and ISDN technologies only.			

The following sections describe the asynchronous outbound call processes, as shown in Figure 11, "Basic Asynchronous Outbound Call State Diagram", on page 54.

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 Overlap Sending

Note: This functionality applies to E1, T1 and ISDN technologies only.

In the Dialing state, if the remote side requests more information such as the destination address, the GCEV_REQMOREINFO event is generated and the call transitions to the SendMoreInfo state. The **gc_SendMoreInfo( )** function is issued to send more information. If the remote side still requests more information, the GCEV_REQMOREINFO event is generated again. Once the remote side has received sufficient information, it indicates that the call is proceeding, and accepts or answers the call. Some technologies, such as ISDN and SS7, do not have any messages or signals to request more information. For such protocols, the application never gets the unsolicited GCEV_REQMOREINFO event. In this case, the application may call the **gc_SendMoreInfo( )** function to send more information as it becomes available.

3.4.2.8 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.

3.4.2.9 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 appropriate Dialogic® Global Call API Technology Guide for that technology.

Figure 12 shows a basic asynchronous call scenario for outgoing calls.

Figure 12. Asynchronous Outbound Call Scenario

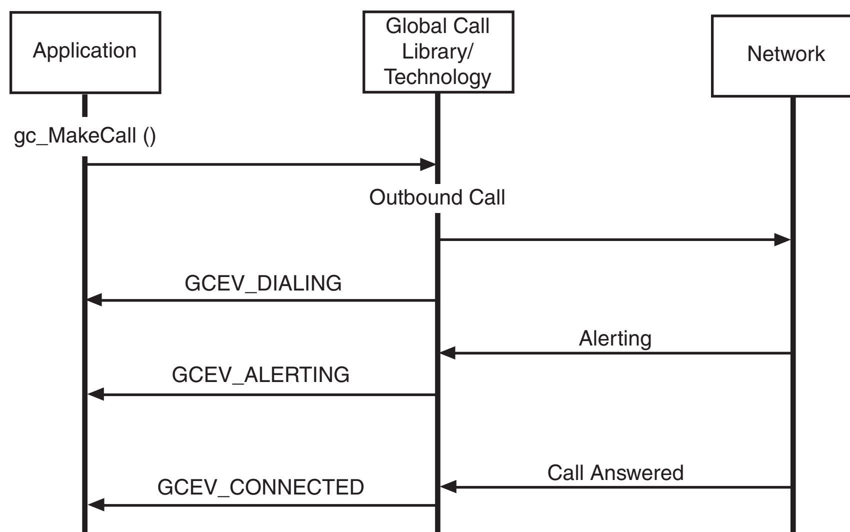


Figure 13 shows an asynchronous call scenario for outgoing calls with call acknowledgement.

Figure 13. Asynchronous Outbound Call Scenario With Call Acknowledgement

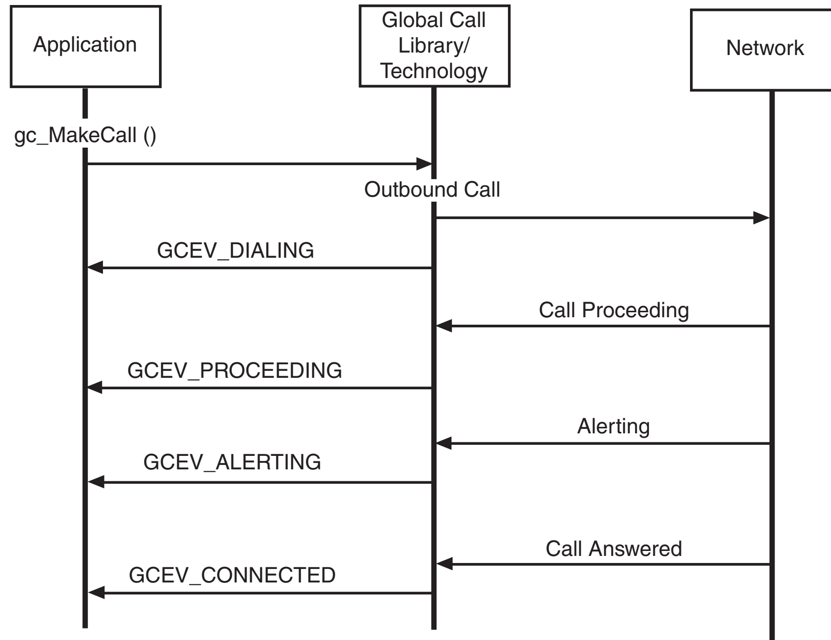
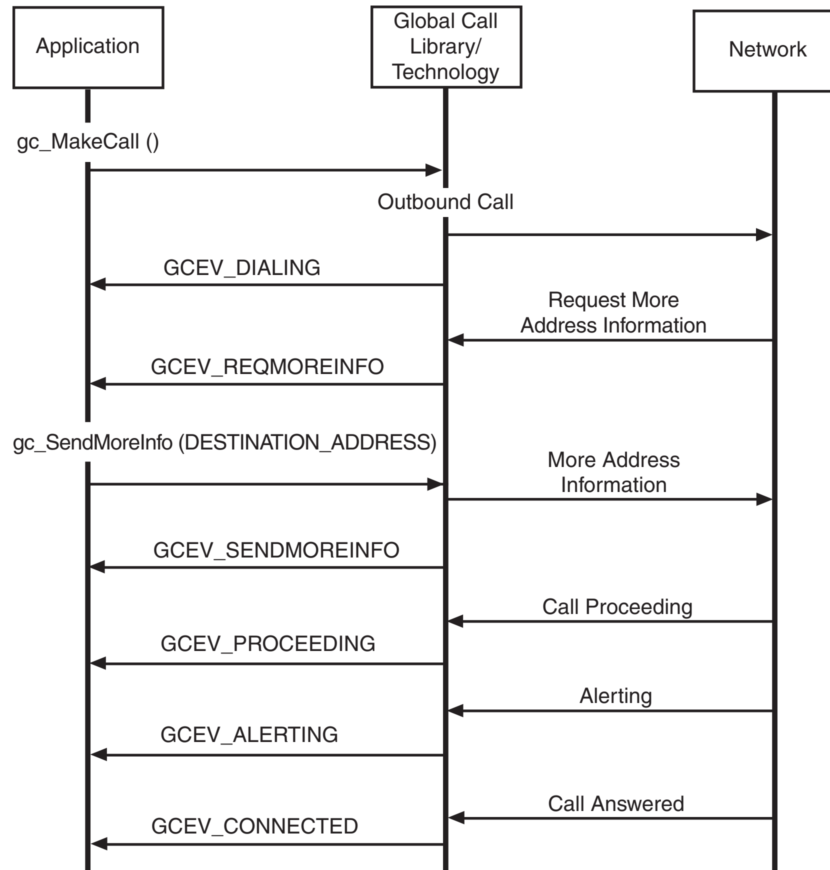


Figure 14 shows an asynchronous call scenario for outgoing calls with overlap sending.

Note: This scenario applies to E1, T1 and ISDN technologies only.

Figure 14. Asynchronous Outbound Call Scenario With Overlap Sending



3.4.3 Call Termination in Asynchronous Mode

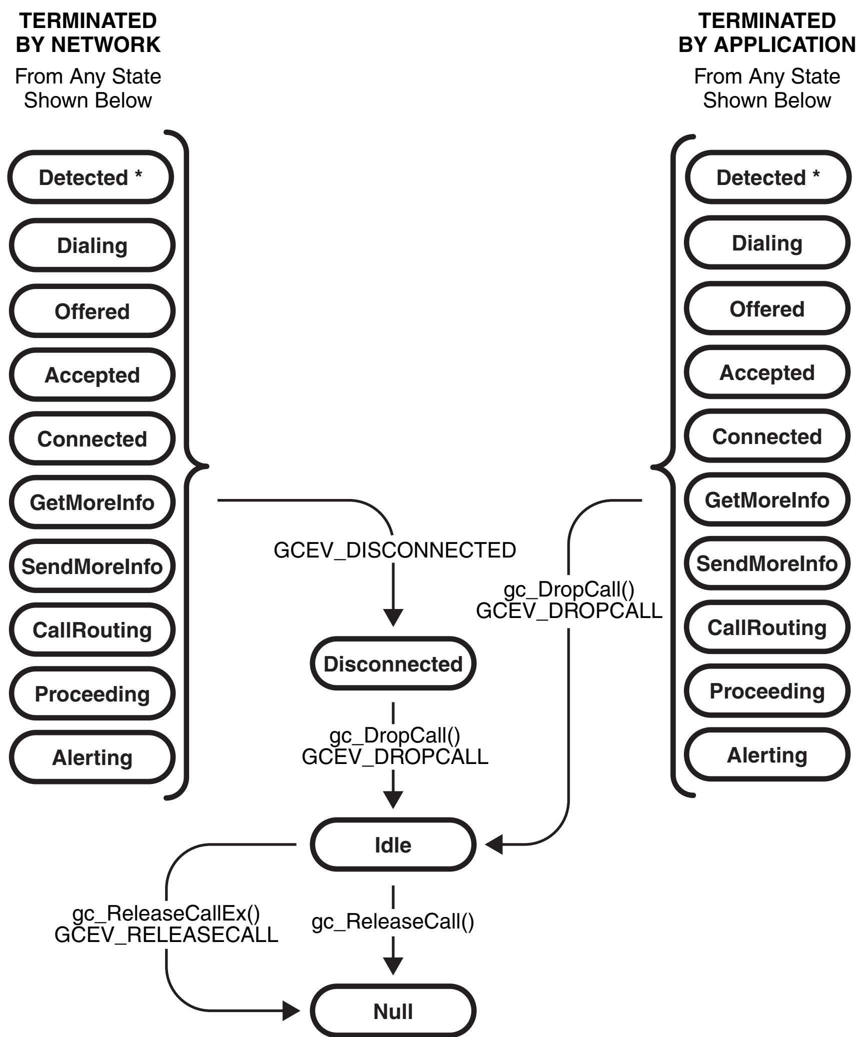
This section describes how calls are terminated and shows call scenarios for asynchronous call termination. The following topics describe call termination in asynchronous mode:

- 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 15 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 15. Asynchronous Call Tear-Down State Diagram



Note: In Figure 15, the GetMoreInfo and SendMoreInfo states apply to E1, T1 and ISDN technologies only.

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, SendMoreInfo †, Proceeding, Alerting, GetMoreInfo †, CallRouting **Next**: GCEV_DROPCALL -> Idle state	**gc_DropCall( )**	GCEV_DROPCALL
Idle (GCST_IDLE) Not Maskable	**Previous**: Offered, Accepted, Connected, Dialing, SendMoreInfo †,Proceeding, Alerting, GetMoreInfo †,CallRouting, Disconnected **Next**: GCEV_RELEASECALL -> Null	**gc_ReleaseCallEx( )**	GCEV_RELEASECALL
† Applies to E1, T1 and ISDN technologies only.			

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 appropriate Dialogic® Global Call API Technology Guide for that technology.

Figure 16 shows an asynchronous user initiated call termination scenario.

Figure 16. User Initiated Asynchronous Call Termination Scenario

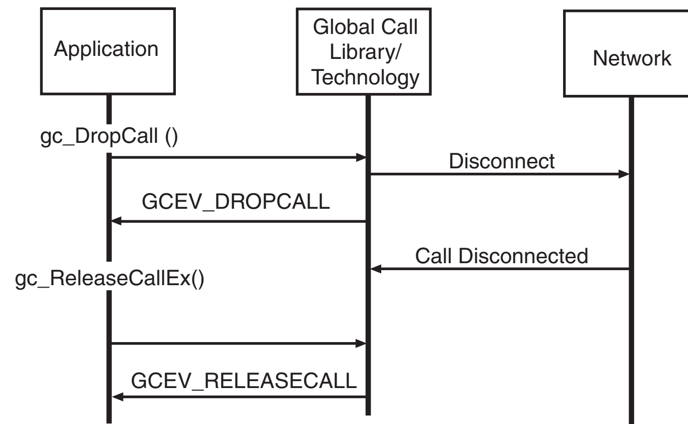
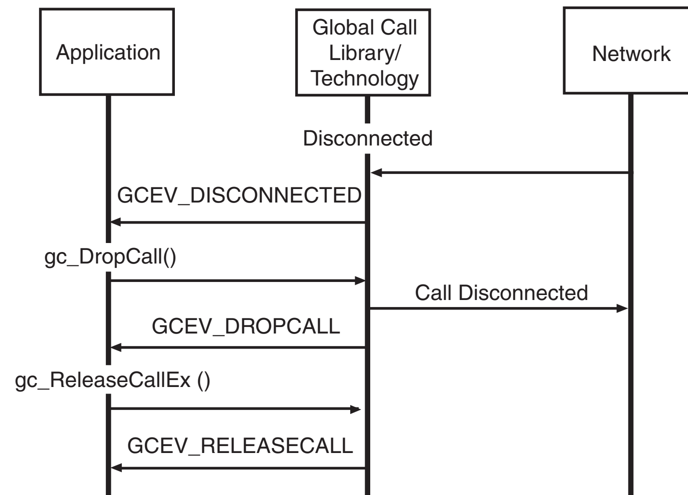


Figure 17 shows an asynchronous network initiated call termination scenario.

Figure 17. 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 technology-specific unsolicited events also require a signal handler (see the appropriate Dialogic® Global Call API 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

3.5 Advanced Call Control with Call Hold and Transfer

Note: The advanced call model applies only to E1, T1 and ISDN technologies. It does not apply to IP technology which uses a different scheme for features such as call transfer. See the *Dialogic® Global Call IP Technology Guide* for more information.

This section describes the advanced call state model. Topics include:

- Advanced Call State Model Overview
- Advanced Call States for Hold and Transfer
- Call Hold
- Call Transfer

3.5.1 Advanced Call State Model Overview

The advanced call model provides additional call control functionality over the basic call model, adding the ability to transfer calls, place calls on hold and retrieve calls on hold. This section provides brief descriptions of the API functions used to hold, retrieve, and transfer calls and

describes the call state transitions that occur when the functions are used. This section also provides figures that illustrate the call state transitions for advanced call model functions.

Note: The hold, retrieve, and transfer functions are supported by particular protocols for the ISDN, E1 (PDKRT only) and T1 (PDKRT only) technologies. For more information, see the function descriptions in the *Dialogic® Global Call API Library Reference* and the appropriate Dialogic® Global Call API Technology Guide.

3.5.2 Advanced Call States for Hold and Transfer

Two advanced call states are appended to the basic call model to support call hold and transfer. These advanced call states are as follows:

On-Hold State (GCST_ONHOLD)

A call must be in the Connected call state to be put on hold. When a call is put on hold, the remote party is often routed via the local switch or network to receive background music while temporarily suspended from conversing with the local party. The call remains on hold until the application retrieves the call, effectively re-transitioning it into the Connected, "conversational" state. The application may not issue a **gc_MakeCall( )** or receive another call while a call is in the On-hold state.

There is no limit to the number of times a call may be placed in and retrieved from the On-hold state. In addition, either the called party or the calling party can put the call in the On-hold state.

The On-hold call state applies only to call scenarios where a single call is present on the specified channel. The On-hold call state does not apply to call transfer scenarios that use the On-Hold Pending Transfer call state instead.

On-Hold Pending Transfer State (GCST_ONHOLDPENDINGTRANSFER)

During a supervised call transfer, two calls are made accessible to the local channel. Both calls must be in the Connected call state. The call that is temporarily suspended from conversing is considered to be in the On-hold Pending Transfer call state. This call is often routed via the local switch or network to receive background music while awaiting completion of the call transfer.

Both the suspended call and the currently active call may be swapped at any time so that the call that was in the On-hold Pending Transfer state is now actively connected, while the former active call is placed in the On-hold Pending Transfer state. There is no limit to the number of times two calls may be swapped between the On-hold Pending Transfer and "Connected" states. The completion of the call transfer is independent of which call is active or on hold.

3.5.3 Call Hold

The advanced call model allows the application to place a call on hold. The Dialogic® Global Call API provides the following functions to place a call on hold and, subsequently, to retrieve the call on hold:

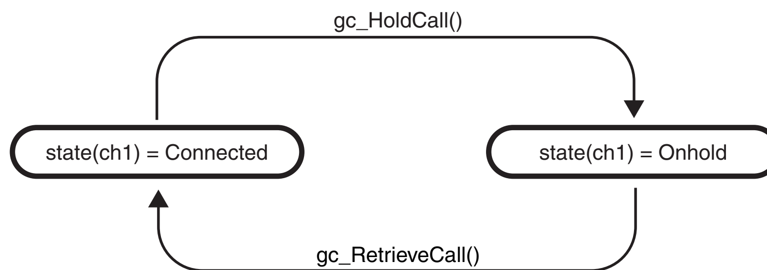
gc_HoldCall( )
place a call on hold

gc_RetrieveCall( )
retrieve a call from hold

The **gc_HoldCall( )** function places an active call in the On-hold (GCST_ONHOLD) state. The **gc_RetrieveCall( )** function retrieves the call from the GCST_ONHOLD state and returns it to the Connected (GCST_CONNECTED) state.

Figure 18 illustrates the transition between call states when a call is put on hold and then retrieved.

Figure 18. Call State Transitions for Hold and Retrieve



Calls in the On-hold state must be returned to the Connected state before they can be dropped. Calls are dropped following the Basic Call scenario. See Section 3.4, "Basic Call Control in Asynchronous Mode" for more information.

3.5.4 Call Transfer

This section describes the different types of call transfer. Topics include:

- Call Transfer Overview
- Supervised Transfers
- Unsupervised Transfers

3.5.4.1 Call Transfer Overview

There are two types of call transfers:

Supervised transfers
: 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

Unsupervised transfers
: the call is sent without any consultation or announcement by the person transferring the call. Unsupervised transfers are also known as one-step transfers or blind transfers

Note: The the call transfer implementations described in this section are common to a number of different technologies. However, not all technologies support these implementations and some technologies have technology-specific implementations for call transfer. See the appropriate Dialogic® Global Call Technology Guide for technology-specific information on call transfer.

Supervised transfers use the following Dialogic® Global Call API functions:

gc_SetupTransfer( )
: initiates a supervised transfer

gc_CompleteTransfer( )
: completes a supervised transfer

gc_SwapHold( )
: switches between the consultation call and the call pending transfer

Unsupervised transfers use the following Dialogic® Global Call API function:

gc_BlindTransfer( )
: initiates and completes an unsupervised (one-step) transfer

3.5.4.2 Supervised Transfers

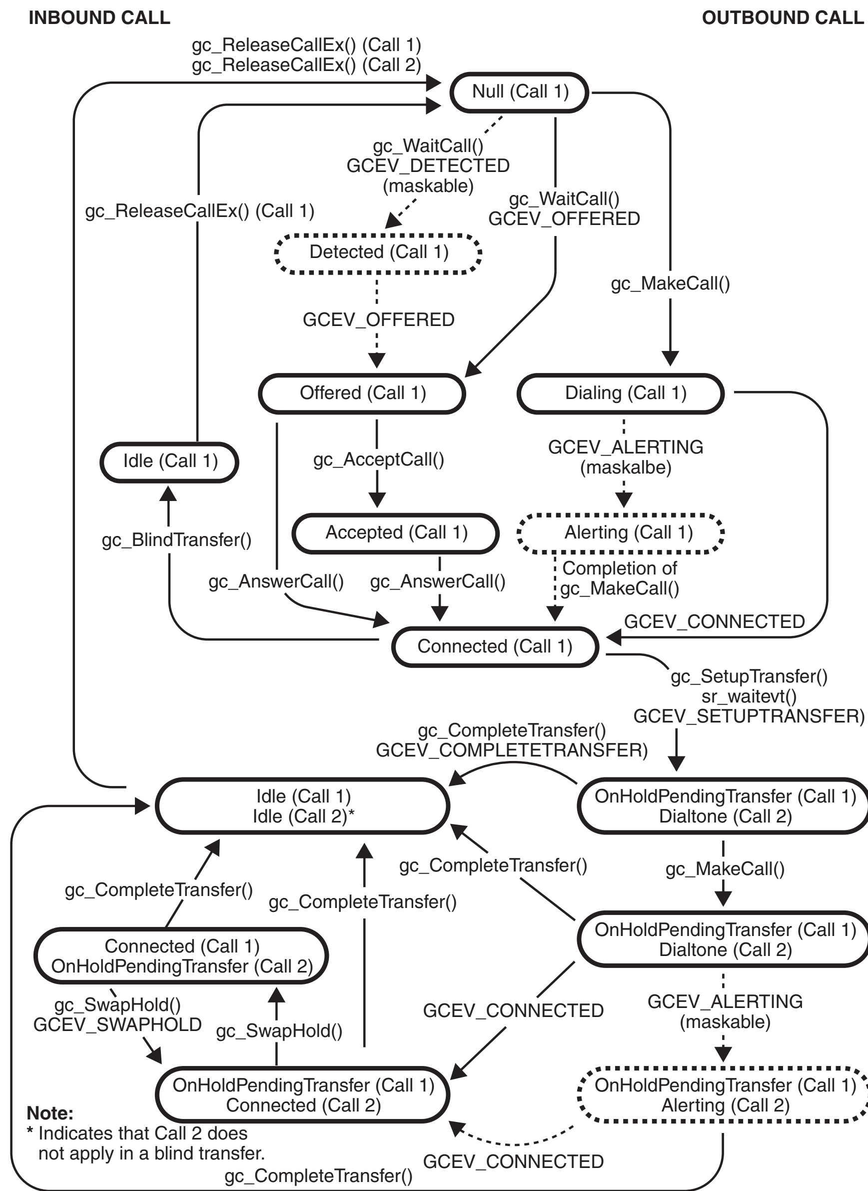
A supervised transfer begins with a successful call to the **gc_SetupTransfer( )** function. The following steps describe how the transfer is completed:

1. Successful call to the **gc_SetupTransfer( )** function changes the state of the original call to the GCST_ONHOLDPENDINGTRANSFER state.
2. A consultation CRN is allocated with the initial state of GCST_DIALTONE and is returned by the **gc_SetupTransfer( )** function.
3. The **gc_MakeCall( )** function is called to establish a connection on the consultation call. The CRN returned by **gc_MakeCall( )** is the same CRN as was returned by **gc_SetupTransfer( )**.
4. The consultation call proceeds similarly to a singular outbound call proceeding through the GCST_DIALING and GCST_ALERTING (if enabled) call states. (See Section 3.4, "Basic Call Control in Asynchronous Mode" for more information.)
5. If the consultation call is successfully established, the state of the consultation call changes to the GCST_CONNECTED state, and the state of the original call remains unchanged.
6. While the consultation call is in the GCST_CONNECTED state, the **gc_SwapHold( )** function may be used to switch between the call pending transfer and the consultation call.
7. A call to the **gc_CompleteTransfer( )** function transfers the original call to the consultation call and internally drops both channels.
8. The states of the original and the consultation call both change to the GCST_IDLE state upon receipt of the GCEV_COMPLETETRANSFER event.
9. The application must call **gc_ReleaseCallEx( )** for both of the calls to release the resources allocated for both channels.

Note: The consultation call may be terminated at any point in the process by the application or by the detection of a call disconnect from the network.

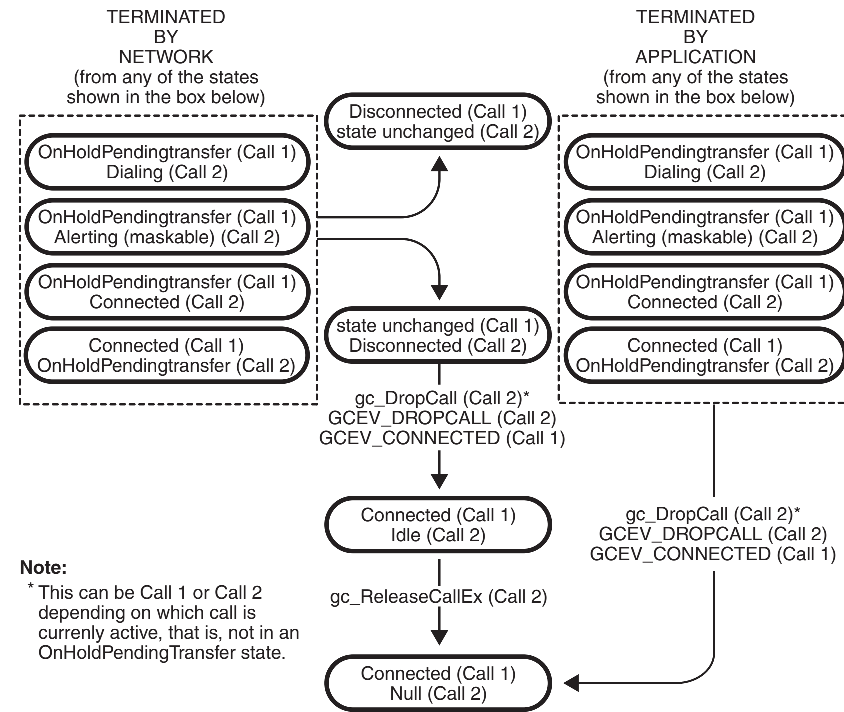
The call state transitions that occur during a supervised transfer are shown in Figure 19 (which also shows the call state transitions for an unsupervised transfer).

Figure 19. Call State Model for Supervised and Unsupervised Transfers



If the network or application terminates a call during a transfer, the call state transitions are as shown in Figure 20.

Figure 20. Call Termination by the Network or Application During a Transfer



Note: In Figure 20, when **gc_DropCall( )** is issued, an unsolicited GCEV_CONNECTED event is received for call 1 transitioning it back to the Connected state.

3.5.4.3 Unsupervised Transfers

In an unsupervised transfer, a successful call to the **gc_BlindTransfer( )** function transfers the call in a single step, without any consultation or announcement by the person transferring the call. Internally, the currently connected call is placed on hold, the new party is dialed, and, finally, the connection to both parties is relinquished. When the application receives the GCEV_BLINDTRANSFER event, the original call enters the GCST_IDLE state. At this point the application must call **gc_ReleaseCallEx( )** for the call to release the allocated resources.

Once the new party is dialed, the control and responsibility for the results of the transfer, whether successfully connected or not, lie totally with the remote party once the transfer is relinquished. Only one call is controlled by the application as the transfer is initiated internally via the protocol.

Unsupervised transfers do not provide call progress results for the transfer nor do they support terminating the transfer at any point via the **gc_DropCall( )** function.

Figure 19 illustrates the call state transitions that occur in an unsupervised transfer, which basically includes only:

- The transition of Call 1 from the Connected to the Idle state (invoked by the **gc_BlindTransfer( )** function)
- The transition of Call 1 from the Idle to the Null state (invoked by the **gc_ReleaseCallEx( )** function).

Event Handling 4

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

- Overview of Event Handling 71
- Event Categories 71
- Blocked and Unblocked Event Handling. 72
- Event Retrieval. 73
- Events Indicating Errors 74
- Masking Events 74
- Event Handlers. 74

4.1 Overview of Event Handling

The Dialogic® Global Call API 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 Dialogic® Global Call API event categories is provided in this chapter. Specific event definitions are described in the *Dialogic® Global Call API Library Reference*. See the appropriate Dialogic® Global Call API Technology Guide for technology-specific event information.

4.2 Event Categories

The events that can occur when using the Dialogic® 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 *Dialogic® Global Call API Library Reference* for detailed information about each event and the appropriate Dialogic® Global Call API Technology Guide for any technology-specific event information.

4.3 Blocked and Unblocked Event Handling

Dialogic® Global Call API 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_ReleaseCall( )** (applies to E1, T1 and ISDN technologies only)
- **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( )**, and **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 Dialogic® Global Call API 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 99.

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 Dialogic® Standard Runtime Library (SRL) event retrieval mechanisms (see the *Dialogic® 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 Dialogic® Global Call API events, the structure contains Dialogic® Global Call API related information (CRN and line device) used by the application. For events that are not Dialogic® Global Call API 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 events 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 Dialogic® Global Call API and additional reason information provided by the underlying call control library. See the *Dialogic® Global Call API Library Reference* for more information.

4.6 Masking Events

Some events are maskable. See the **gc_SetConfigData( )** function description in the *Dialogic® 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. Event handlers are described in the following topics:

- Event Handlers for Linux
- Event Handlers for Windows®

4.7.1 Event Handlers for Linux

The following guidelines apply to event handlers (for detailed information, see the *Dialogic® Standard Runtime Library API Programming Guide*):

- More than one handler can be enabled for an event.
- General handlers can be enabled that handle any event on a specified device.
- Handlers can be enabled to handle any event on any device.
- Synchronous functions cannot be called in a handler.
- Handlers must return a 1 to advise the SRL to keep the event in the SRL queue and a 0 to advise the SRL to remove the event from the SRL queue.

When using the asynchronous with event handlers model, after initiation of the asynchronous function, the process cannot receive termination (solicited) or unsolicited events until the **sr_waitevt( )** function is called. When using this model, the main process typically issues a single call for the **sr_waitevt( )** function. If a handler returns a non-zero value, the **sr_waitevt( )** function returns to the main process.

4.7.2 Event Handlers for Windows®

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 *Dialogic® 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 the Dialogic® Global Call API. Topics include:

- General Programming Tips 77
- Tips for SRL-Related Programming in a Linux Environment. 78
- Tips for Programming Drop and Insert Applications 78
- Using the Dialogic® Global Call API with Dialogic® Digital Network Interface Boards80

5.1 General Programming Tips

The following tips apply when programming with the Dialogic® Global Call API:

- When using Dialogic® Global Call API functions, the application must use the Dialogic® Global Call API handles (that is, the line device ID and CRN) to access Dialogic® Global Call API functions. Do not substitute a network, voice or media device handle for the Dialogic® Global Call API 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, **nr_scroute( )** (for E1, T1 or ISDN technologies only) or **dx_play( )** (all technologies), you must use the **gc_GetResourceH( )** to retrieve the network, voice or media device handle, associated with the specified Dialogic® Global Call API line device. The **gc_GetResourceH( )** function is only needed if the voice or media resource is associated with a Dialogic® Global Call API line device. If a voice resource is not part of the Dialogic® Global Call API 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 Dialogic® Global Call API library, that is, using Dialogic® Global Call API (**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 Linux environments, when programming in synchronous mode, performance may deteriorate as the number of synchronous processes increase due to the increased Linux overhead needed to handle these processes. When programming multichannel applications, asynchronous mode programming is likely to provide better performance.
- 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 Tips for SRL-Related Programming in a Linux Environment

The following tips apply to Dialogic® Standard Runtime Library (SRL)-related programming in a Linux environment:

- When an SRL is in signaling mode (SIGMODE), do not call any synchronous mode Dialogic® Global Call API function (that is, any function whose mode=EV_SYNC) from within a handler registered to the SRL.
- When an SRL is in signaling mode (SIGMODE) and a Dialogic® Global Call API function is issued synchronously (that is, the function mode=EV_SYNC), ensure that the application only enables handlers with the SRL to catch the exceptions, that is, unsolicited events like GCEV_BLOCKED, GCEV_UNBLOCKED or GCEV_DISCONNECTED. Do not enable wildcard handlers to catch all events. If you enable wildcard handlers, the application may receive unexpected events that should not be consumed.

5.3 Tips for Programming Drop and Insert Applications

Note: This section applies to E1, T1 and ISDN technologies only.

For the Dialogic® Global Call API application, signaling is made available to the application as follows:

- Signaling information is passed to the Dialogic® Global Call API application in the form of call control events; for example, line answer is passed as a GCEV_ANSWERED event.
- Signaling, such as line busy, is available to the application as an EGC_BUSY error code or a GCRV_BUSY result value; line no answer is available as an EGC_NOANSWER error code or GCRV_NOANSWER result value.

- Signaling such as a protocol error, an alerting event, a fast busy, an undefined telephone number, or network congestion are all returned to the application as an EGC_BUSY error code or a GCRV_BUSY result value.
- Protocols without acknowledgement, for example, non-backward CAS signaling protocols, generate a GCEV_DISCONNECTED event with an EGC_BUSY error code or a GCRV_BUSY result value when timeout or protocol errors occur during dialing.

For a drop and insert application in which the calling party needs to be notified of the exact status of the called party's line, the following approach may be used:

- Upon receipt of an incoming call from a calling party, issue a **gc_MakeCall( )** function on the outbound line to the called party.
- After dialing completes on the outbound line, the application should drop the dialing resource, turn off call progress, and connect the inbound line to the outbound line so that the calling party can hear the tones returned on the outbound line. These tones provide positive feedback to the calling party as to the status of the called party's line.
- If the status of the called party's line is such that the call cannot be completed, the calling party hangs up and the application can then drop the call and release the resources used. Otherwise, when the call is answered, a GCEV_CONNECTED event will be received.

When call progress is being used, after dialing completes, the call progress software looks for ringback or voice on the outbound line. When ringback is detected, a GCEV_ALERTING event is generated. When voice is detected, a GCEV_ANSWERED event is generated. An unacceptable amount of time may lapse before either of these events is generated while the calling party is waiting for a response that indicates the status of the call. Thus, for drop and insert applications, call progress should be disabled as soon as dialing completes and the inbound and outbound lines connected so as to provide the calling party with immediate outbound line status and voice cut-through.

For a drop and insert application in which a call cannot be completed, the application can simulate and return a busy tone or a fast busy (redial) tone to the calling party. Typically, this condition occurs when a GCEV_DISCONNECTED event is generated due to a timeout or a protocol error during dialing or due to R2 backward signaling indicating a busy called party's line, equipment failure, network congestion or an invalid telephone number.

When a call cannot be completed because the called party's line is busy:

1. Use a tone or voice resource to generate a busy tone (60 ipm [impulses per minute]) or to record a busy tone.
2. Connect the busy tone to the calling party's line or play back the recorded busy tone file.
3. Drop and release the calling party's line when a GCEV_DISCONNECTED event is received.

When a call cannot be completed because of equipment failure, network congestion or an invalid telephone number:

1. Use a tone or voice resource to generate a fast busy tone (120 ipm) or to record a fast busy tone.
2. Connect the fast busy tone to the calling party's line or play back the recorded fast busy tone file.

3. Drop and release the calling party's line when a GCEV_DISCONNECTED event is received.

For voice function information, see the *Dialogic® Voice API Library Reference* for your operating system.

5.4 Using the Dialogic® Global Call API with Dialogic® Digital Network Interface Boards

The Dialogic® Host Media Processing (HMP) Software can be used in conjunction with Dialogic® Digital Network Interface boards that provide physical E1 and T1 interfaces. These Digital Network Interface boards are based on the Dialogic® DM3 architecture. The Dialogic® Global Call API supports the development of applications that use these boards. The following topics provide guidelines for using the Dialogic® Global Call API with Dialogic® Digital Network Interface boards:

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

5.4.1 Routing Overview

The Dialogic® 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.

5.4.2 Working with Flexible Routing Configurations

The following topics provide more information about using the Dialogic® Global Call API with Dialogic® Digital Network Interface boards that use the flexible routing configuration:

- Determining Channel Capabilities (Flexible Routing)
- Using Device Handles (Flexible Routing)
- Multi-Threading and Multi-Processing (Flexible Routing)
- Initializing an Application (Flexible Routing)
- Initializing Dialogic® Global Call API (Flexible Routing)
- Device Initialization Hint (Flexible Routing)
- Using Protocols (Flexible Routing)

5.4.2.1 Determining Channel Capabilities (Flexible Routing)

Dialogic® Digital Network Interface boards support three different types of voice devices:

- E1 CAS compatible
- T1 CAS compatible
- ISDN compatible

The E1 CAS compatible is a superset of T1 CAS compatible, and the T1 CAS compatible is a superset of ISDN compatible.

When using the Dialogic® Global Call API, only certain Dialogic® DM3 network interface devices can be associated with certain other Dialogic® DM3 voice devices using **gc_OpenEx( )** or **gc_AttachResource( )**. Attaching Dialogic® DM3 devices together depends on the network protocol used and voice device capabilities. Specifically:

- A Dialogic® DM3 ISDN network device can be attached to any Dialogic® DM3 voice device.
- A Dialogic® DM3 T1 CAS network device must be attached to a T1 CAS compatible Dialogic® DM3 voice device.
- A Dialogic® DM3 E1 CAS network device must be attached to an E1 CAS compatible Dialogic® DM3 voice device.

An application can query the capabilities of a device using the **dx_getfeaturelist( )** function which includes information about the front end supported, meaning ISDN, TI CAS, or R2/MF. See the *Dialogic® Voice API Library Reference* for more information about the **dx_getfeaturelist( )** function.

When using Dialogic® Global Call API, if a voice device is not CAS or R2/MF capable, it cannot be attached (either in the **gc_OpenEx( )** function or when using the **gc_AttachResource( )** function) to a network interface device that has CAS or R2/MF loaded. Likewise, if a voice device is not routable, it cannot be used in a **gc_AttachResource( )** call.

While a network interface protocol cannot be determined programmatically, the **dx_getfeaturelist( )** function provides a programmatic way of determining voice capability so that the application can make decisions.

5.4.2.2 Using Device Handles (Flexible Routing)

When using Dialogic® Digital Network Interface boards, application performance may be a consideration when opening and closing devices using Dialogic® Global Call API. If an application must use Dialogic® Global Call API 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.

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

When using Dialogic® Digital Network Interface boards, 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.

5.4.2.4 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 Dialogic® Global Call API for call control (that is, for call setup and tear-down), all Dialogic® network interface devices must be opened using the **gc_OpenEx( )** function.

Note: When call control is not required, such as with ISDN NFAS, **dt_open( )** can be used to open Dialogic® DM3 network interface devices.

Once the call has been established, voice and or data streaming should be done using the Dialogic® 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.

5.4.2.5 Initializing Dialogic® Global Call API (Flexible Routing)

This scenario is one where an application uses Dialogic® Digital Network Interface boards in a flexible routing configuration. When initializing an application to use boards based on the Dialogic® DM3 architecture, you must use Dialogic® Global Call API to handle the call control.

Take note of the following flexibility that exists for the **gc_OpenEx( )** function when opening a Dialogic® Global Call API line device on Dialogic® Digital Network Interface boards:

- Due to the nature of the Dialogic® DM3 architecture, the protocol name is irrelevant at the time of opening the Dialogic® Global Call API line device; that is, the protocol name is ignored. Also, when using R4 with boards based on the Dialogic® DM3 architecture, all protocols are bi-directional. You do not need to dynamically open and close devices to change the direction of the protocol.
- It is not necessary to specify a voice device name when opening a Dialogic® Global Call API line device. If you specify the voice device name, the network interface device is automatically associated with the voice device (they are attached and routed on the TDM bus). If you do not specify the voice device name when you open the Dialogic® Global Call API line device, you can separately open a voice device, and then attach and route it to the network interface device.

  For boards that use the Dialogic® DM3 architecture in a flexible routing configuration, only the network device name is required.

The following procedure shows how to initialize Dialogic® Global Call API when using Dialogic® Digital Network Interface boards.

Note: In Windows®, use the **sr_getboardcnt( )** function with the class name set to DEV_CLASS_DTI and DEV_CLASS_VOICE to determine the number of network and voice boards in the system, respectively. In Linux, use SRL device mapper functions to return information about the structure of the system. For information on these functions, see the *Dialogic® Standard Runtime Library API Library Reference*.

1. Start/Initialize Dialogic® Global Call API using **gc_Start( )**.
2. Use **gc_OpenEx( )** to open a Dialogic® Global Call API line device.

- Specify the network interface device name and the protocol name in the **devicename** parameter, as in the following example:
  `":N_dtiB1T1:P_ISDN"`
- Alternatively, specify the network interface device name, the voice device name, and the protocol name in the **devicename** parameter, as in the following example:
  `":N_dtiB1T1:V_dxxxB1C1:P_ar_r2_io"`

3. Obtain the voice channel device handle.
   - Open a voice channel device (for example, dxxxB1C1) with **dx_open( )** to get its handle.
   - Alternatively, if you specified the voice device name in the **devicename** parameter in step 2, use **gc_GetResourceH( )**, with a **resourcetype** of GC_VOICEDEVICE, to get the handle.
4. Attach the voice and network interface devices.
   - Use **gc_AttachResource( )** to attach the voice resource and the network interface line device.
   - Alternatively, if you specified the voice device name in the **devicename** parameter in step 2, the voice and network interface devices are attached by nature of the **gc_OpenEx( )**, so no action is necessary for this step.
5. Use **gc_GetResourceH( )**, with a **resourcetype** of GC_NETWORKDEVICE, to obtain the network interface time slot device handle that is associated with the line device.
6. Set up TDM bus full duplex routing between the network interface device and voice device.
   - Use **nr_scroute**(FULL DUPLEX).
   - Alternatively, if you specified the voice device name in the devicename parameter in step 2, the network interface device and voice device are automatically routed on the TDM bus by nature of the **gc_OpenEx( )**.

Repeat steps 2 to 6 for all Dialogic® Global Call API device line devices.

5.4.2.6 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.

5.4.2.7 Using Protocols (Flexible Routing)

For ISDN protocols, the protocol to use is determined at board initialization time and not when opening a Dialogic® Global Call API device. Protocol parameters are configured in the CONFIG

file before the firmware is downloaded to the board. If a protocol is specified in the **devicename** parameter of the **gc_OpenEx( )** function when opening a device, it is ignored.

For T1/E1 CAS/R2MF protocols, the protocol to use for a trunk is selected using the "Trunk Configurator" feature of the Dialogic® Configuration Manager (DCM). Protocol files are provided with the system software in the *\data* directory under the Dialogic home directory. A protocol can be configured by changing the parameter values in the corresponding Country Dependent Parameter (CDP) file located in the *\data* directory. See the *Dialogic® Global Call Country Dependent Parameters (CDP) for PDK Protocols Configuration Guide* for details on the parameters that can be changed for each protocol. If a protocol is specified in the **devicename** parameter of the **gc_OpenEx( )** function when opening a device, it is ignored.

5.4.3 Handling Multiple Call Objects Per Channel in a Glare Condition

When using Dialogic® Digital Network Interface boards, Dialogic® Global Call API 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 Dialogic® Global Call API library does not wait for the outbound call to be released before notifying the application of the inbound call.

Table 10. Handling Glare

Application	Dialogic® Global Call API 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 the Dialogic® Global Call API. Topics include the following:

- Error Handling Overview 85

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 Dialogic® 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 Dialogic® Global Call API and call control library information. To retrieve the information, this function must be called immediately after the Dialogic® Global Call API function failed. This function returns a result value associated directly with the Dialogic® Global Call API and call control library.

The **gc_ResultInfo( )** function retrieves information about solicited and unsolicited events when a Dialogic® Global Call API application gets an expected or unexpected event. To retrieve the information, the **gc_ResultInfo( )** function must be called immediately after a Dialogic® Global Call API event arrives and before the next event returns Dialogic® Global Call API and call control library information related to the last Dialogic® Global Call API 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 appropriate Dialogic® Global Call API 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_GETCONFIGDATA_FAIL (E1, T1 and ISDN technologies only), or GCEV_SETCONFIGDATA_FAIL (all technologies) 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 60. 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 Dialogic® Global Call API reasons (error code or result values) or a call control library-specific reason (see the appropriate Dialogic® Global Call API Technology Guide).

Call Control 7

This chapter describes the Dialogic® Global Call API capabilities relating to call control. Topics include:

- Call Progress Analysis when Using IP Technology 87
- Call Progress Analysis when Using Dialogic® Digital Network Interface Boards . . . 87
- Resource Routing 93
- Feature Transparency and Extension 93

7.1 Call Progress Analysis when Using IP Technology

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

7.2 Call Progress Analysis when Using Dialogic® Digital Network Interface Boards

Note: This section applies to E1, T1 and ISDN technologies only.

When using Dialogic® Digital Network Interface boards, the Dialogic® Global Call API provides a consistent method of pre-connect call progress and post-connect call analysis across E1/T1 CAS, and ISDN protocols. The level of support that the Dialogic® Global Call API provides is described in the following topics:

- Call Progress Analysis Definition
- Configuring Default Call Progress Analysis Parameters
- Configuring Call Progress Analysis on a Per Call Basis
- Setting Call Analysis Attributes on a Per Call Basis
- Configuring Call Progress Analysis on a Per Channel Basis
- Setting Call Analysis Attributes on a Per Channel Basis
- Customizing Call Progress Tones on a Per Board Basis

7.2.1 Call Progress Analysis Definition

Pre-connect call progress determines the status of a call connection, that is, busy, no dial tone, no ringback, etc., and can also include the frequency detection of Special Information Tones (SIT), such as an operator intercept. Post-connect call analysis determines the destination party's media type, that is, voice, fax, or answering machine. The term *call progress analysis* (CPA) is used to refer to call progress and call analysis collectively.

7.2.2 Configuring Default Call Progress Analysis Parameters

Call Progress Analysis (CPA) is characterized by parameters such as **CaSignalTimeout** (the maximum time to wait to detect a call progress tone), **CaAnswerTimeout** (the maximum time that call analysis will wait for ringback to stop), and others that define CPA behavior. Depending on the technology you are using, the default values of CPA parameters may be configurable in the CONFIG file corresponding to the board. If this is the case, the required information is documented in the corresponding Dialogic® Global Call Technology Guide.

Note: When a voice resource has been attached (using either **gc_OpenEx( )** or **gc_AttachResource( )**), by default, the DM3 host runtime library enables the detection of BUSY, RINGING, and SIT tone (that is, pre-connect call progress), even if Call Progress Analysis (CPA) is disabled in the CONFIG file. A user who does not want pre-connect call progress, must explicitly use the **gc_SetConfigData( )** function to disable CPA on that line device. Alternatively, the user can attach the voice resource after the call is connected.

7.2.3 Configuring Call Progress Analysis on a Per Call Basis

To specify call progress analysis behavior, use the **gc_MakeCall( )** function with an associated GC_PARM_BLK (accessible via the GC_MAKECALL_BLK and GCLIB_MAKECALL_BLK structures) containing the CCSET_CALLANALYSIS parameter set ID and the CCPARM_CA_MODE parameter ID with one or more of the following bitmask values ORed together:

GC_CA_BUSY
: Pre-connect busy tone detection

GC_CA_RINGING
: Pre-connect ringback tone detection

GC_CA_SIT
: Pre-connect special information tone (SIT) detection

GC_CA_FAX
: Post-connect fax detection

GC_CA_PVD
: Post-connect positive voice detection (PVD)

GC_CA_PAMD
: Post-connect positive answering machine detection (PAMD)

While the CCPARM_CA_MODE bitmask offers complete flexibility in terms of the selected options, not all option combinations make sense. For this reason, the following defines, that can also be used as values to the CCPARM_CA_MODE parameter ID, identify the most logical and traditionally used option combinations:

GC_CA_DISABLE
: Call progress and call analysis disabled

GC_CA_PREONLY
: Busy and Ringing enabled

GC_CA_PREONLY_SIT
Busy, Ringing and SIT enabled

GC_CA_POSTONLY_PVD
Fax and PVD enabled

GC_CA_POSTONLY_PVD_PAMD
Fax, PVD and PAMD enabled

GC_CA_ENABLE_PVD
Busy, Ringing, and SIT enabled; fax and PVD enabled

GC_CA_ENABLE_ALL
Busy, Ringing, and SIT enabled; fax, PVD and PAMD enabled

These options correspond closely to call progress and call analysis options available when using the Dialogic® Voice API as indicated in Table 11. See the "Call Progress Analysis" chapter in the *Dialogic® Voice API Programming Guide*.

Table 11. Call Progress Analysis Settings and Possible Results

CCPARM_CA_MODE Setting	Equivalent ca_intflg Setting in DX_CAP Structure When Using Voice API
GC_CA_DISABLE	DISABLE
GC_CA_PREONLY	DX_OPTDIS
GC_CA_PREONLY_SIT	DX_OPTNOCON or DX_OPTEN
GC_CA_POSTONLY_PVD	DX_PVDENABLE
GC_CA_POSTONLY_PVD_PAMD	DX_PAMDENABLE
GC_CA_ENABLE_PVD	DX_PVDOPTNOCON or DX_PVDOPTEN
GC_CA_ENABLE_ALL	DX_PAMDOPTEN

When an option that enables call progress is selected, a GCEV_DISCONNECTED event can be received. The **gc_ResultInfo( )** function can be used to get more information about the event. The possible cause values (the gcValue field in the associated GC_INFO structure) that can be retrieved are:

GCRV_BUSY
Busy

GCRV_NOANSWER
No Answer

GCRV_CEPT
SIT, Operator Intercept

GCRV_UNALLOCATED
SIT, Vacant Circuit, non-registered number

GCRV_CONGESTION
SIT, No Circuit Found
or
SIT, Reorder, system busy

See the *Dialogic® Global Call API Library Reference* for more information about the **gc_ResultInfo( )** function.

When an option that enables call analysis is selected, a GCEV_MEDIADETECTED event can be received. The **gc_GetCallInfo( )** function can be used to determine the type of detection (by setting the **info_id** function parameter to CONNECT_TYPE). The **valuep** function parameter indicates the connect type when the function completes. Typical values in this context are:

GCCT_FAX
: Fax detection

GCCT_PVD
: Positive voice detection (PVD)

GCCT_PAMD
: Positive answering machine detection (PAMD)

See the *Dialogic® Global Call API Library Reference* for more information about the **gc_GetCallInfo( )** function.

7.2.4 Setting Call Analysis Attributes on a Per Call Basis

Certain call analysis attributes can be configured on a per call basis using the **gc_MakeCall( )** function with an associated GC_PARM_BLK (accessible via the GC_MAKECALL_BLK and GCLIB_MAKECALL_BLK structures) that contains the CCSET_CALLANALYSIS parameter set ID and one of the following parameter IDs:

CCPARM_CA_PAMDSPDVAL
: Positive answering machine detection (PAMD) speed value. Quick or full evaluation of answering machine detection. Possible values are:
  - PAMD_FULL – Full evaluation of response.
  - PAMD_QUICK – Quick look at connection characteristics.
  - PAMD_ACCU – Recommended setting. Does the most accurate evaluation detecting live voice as accurately as PAMD_FULL, but is more accurate than PAMD_FULL (although slightly slower) in detecting an answering machine. Use PAMD_ACCU when accuracy is more important than speed. This is the default value.

CCPARM_CA_NOANSR
: No Answer. The length of time (in 10 ms units) to wait after the first ringback before deciding that the call is not answered. Possible values are in the range 0 to 65535. The default value is 3000.

CCPARM_CA_NOSIG
: Continuous No Signal. The maximum amount of silence (in 10 ms units) allowed immediately after cadence detection begins. If exceeded, a *no ringback* is returned. Possible values are in the range 0 to 65535. The default value is 4000.

CCPARM_CA_PAMDFAILURE
: PAMD Fail Time. The maximum time (in 10 ms units) to wait for positive answering machine detection (PAMD) or positive voice detection (PVD) after a cadence break. Possible values are in the range 0 to 65535. The default value is 800.

CCPARM_CA_PAMD_QTEMP

PAMD Qualification Template. Specifies which PAMD template to use. Possible values are:

- PAMD_QUAL1TMP – First predefined qualification template. This is the default value.
- -1 – No qualification template

Setting CCPARM_CA_PAMD_QTEMP to a value of PAMD_QUAL2TMP is **not** supported.

Note: The CCPARM_CA_PAMD_QTEMP parameter can also be set to a qualification template ID that is defined in the CONFIG file.

CCPARM_CA_PVD_QTEMP

PVD Qualification Template. Specifies which PVD template to use. Possible values are:

- PAMD_QUAL1TMP – First predefined qualification template. This is the default value.
- -1 – No qualification template

Setting CCPARM_CA_PVD_QTEMP to a value of PAMD_QUAL2TMP is **not** supported.

Note: The CCPARM_CA_PVD_QTEMP parameter can also be set to a qualification template ID that is defined in the CONFIG file.

By default, qualification template parameters are set to the most common values. However, it is possible to tune these parameters in the CONFIG file as described in a Technical Note available at the following web site: http://www.dialogic.com/support/helpweb/dxall/tnotes/legacy/2000/tn030.htm. The technical note is not written specifically for Dialogic® Host Media Processing (HMP) Software, but the same principle applies.

Note: Dialogic® DM/IP boards use a slightly different version of the PVD/PAMD qualification templates; the values are adjusted for gain loss. CONFIG files for Dialogic® DM/IP boards do include PVD/PAMD qualification templates.

7.2.5 Configuring Call Progress Analysis on a Per Channel Basis

The Dialogic® Global Call API also supports the setting of call progress analysis parameters on a per channel basis. When call progress analysis parameters are set on a per channel basis, the parameter settings apply to all calls made on that channel (line device).

To specify call progress analysis behavior on a per channel basis, use the **gc_SetConfigData( )** function. The relevant function parameters and values in this context are:

target_type

GCTGT_CCLIB_CHAN

target_id

the line device

target_datap

a pointer to a GC_PARM_BLK structure that contains the following parameter set ID and parameter IDs:

- **SetId** – CCSET_CALLANALYSIS
- **ParmId** – CCPARM_CA_MODE that can take any of the values described in Section 7.2.3, “Configuring Call Progress Analysis on a Per Call Basis”, on page 88.

In earlier releases, when using CAS PDK protocols, it was possible to specify call progress and call analysis on a per channel basis using the **gc_SetParm( )** function to enable or disable the GCPR_CALLPROGRESS and GCPR_MEDIADETECT parameters. See the *Dialogic® Global Call E1/T1 CAS/R2 Technology User's Guide* for more information.

Table 12 shows how the CCPARM_CA_MODE values correspond to the GCPR_CALLPROGRESS and GCPR_MEDIADETECT parameters. This table is provided as a convenience for users that have previously used the **gc_SetParm( )** method and now wish to use the greater flexibility provided by **gc_MakeCall( )** with the CCPARM_CA_MODE parameter.

Table 12. Comparison with Call Progress Analysis Using gc_SetParm( )

GCPR_CALLPROGRESS	GCPR_MEDIADETECT	Equivalent CCPARM_CA_MODE Value
GCPV_DISABLE	GCPV_DISABLE	GC_CA_DISABLE
GCPV_DISABLE	GCPV_ENABLE	GC_CA_POSTONLY_PVD_PAMD
GCPV_ENABLE	GCPV_DISABLE	GC_CA_PREONLY_SIT
GCPV_ENABLE	GCPV_ENABLE	GC_CA_ENABLE_ALL

Note: The **gc_SetConfigData( )** method of setting call progress analysis on a per channel basis is an enhancement over using the **gc_SetParm( )** with the GCPR_MEDIADETECT and/or GCPR_CALLPROGRESS parameters. Applications should not use both the **gc_SetConfigData( )** method and the **gc_SetParm( )** method on the same line device. If both methods are used, the **gc_SetConfigData( )** method takes precedence.

7.2.6 Setting Call Analysis Attributes on a Per Channel Basis

In addition to enabling and disabling call progress analysis on a per channel basis, certain call analysis attributes can be configured on a per channel basis using the **gc_SetConfigData( )** function. The relevant function parameter values in this context are:

target_type
GCTGT_CCLIB_CHAN

target_id
the line device

target_datap
a pointer to a GC_PARM_BLK structure that contains the following parameter set ID and parameter IDs:

- **SetId** – CCSET_CALLANALYSIS
- **ParmId** – Any of the values described in Section 7.2.4, "Setting Call Analysis Attributes on a Per Call Basis", on page 90.

7.2.7 Customizing Call Progress Tones on a Per Board Basis

When using Dialogic® Digital Network Interface boards, an application can create, delete and query call progress tones on a per board device basis using the **dx_createtone( )**, **dx_deletetone( )**, and **dx_querytone( )** functions and the associated TONE_DATA structure in the Voice API. See the *Dialogic® Voice API Programming Guide* for more information.

7.3 Resource Routing

The Global Call routing functions use the device handles of resources such as a voice channel, a media resource, or a network time slot. The **gc_GetResourceH( )** function can be used to obtain the network, media and voice device handles, 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_NETWORKDEVICE returns the network 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 Dialogic® Global Call Technology Guide for technology-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 Dialogic® Global Call API line device.

7.4 Feature Transparency and Extension

Dialogic® Global Call API 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, “Dialogic® Global Call API Architecture for IP Technology”, on page 18 and Figure 2, “Dialogic® Global Call API Architecture for E1/T1 and ISDN Technologies”, on page 19). FTE is described in the following topics:

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

7.4.1 Feature Transparency and Extension Overview

Feature Transparency and Extension (FTE) is comprised of a number of Dialogic® Global Call API functions. These functions provide the flexibility to extend the generic Dialogic® 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 Dialogic® Global Call API library interface, thereby alleviating the need to access these call control libraries directly via additional APIs.

The Dialogic® Global Call API functions provided for FTE are:

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

gc_GetUserInfo( ) (for E1, T1 and ISDN technologies only)
retrieves technology-specific user information for the specified line device

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.

7.4.2 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 Dialogic® Global Call API function utilizes an extension function identifier (**ext_id**) to specify the feature. The associated technology's Global Call Technology Guide for each call control library lists all the supported extension function identifiers (**ext_id** values) and the associated features that are accessible via the **gc_Extension( )** function (if any).

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

For example, in an ISDN platform the application may use the **gc_Extension( )** function to set D or B channel states. As the concept of B and D channels is ISDN specific and inherently foreign to other protocols, without FTE support, the application would have to link directly with the ISDN call control library then call the required ISDN library functions **cc_SetBChanState( )** or **cc_SetDChanState( )**.

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 Dialogic® Global Call API function call. The reason for this is that the GC_PARM_BLK buffer will be deallocated within Dialogic® Global Call API 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 technology-specific information and is referenced via the extevtdatap pointer in the METAEVENT structure associated with the GCEV_EXTENSIONCMPLT event. See the *Dialogic® 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.

7.4.3 Technology-Specific User Information

The **gc_GetUserInfo( )** (E1, T1 and ISDN technologies only) and **gc_SetUserInfo( )** (all technologies) functions permits the application to retrieve and 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 technology- or protocol-specific, or both. Refer to the associated technology's Dialogic® Global Call API Technology Guide for details on the format of the user information supported and the proper usage of the **gc_GetUserInfo( )** and **gc_SetUserInfo( )** functions.

One typical application of the **gc_SetUserInfo( )** and **gc_GetUserInfo( )** functions is on an ISDN platform where it is desired to transmit and receive user-to-user information elements in each incoming and outgoing message.

In the case of **gc_SetUserInfo( )**, user information is transmitted to the remote side embedded in a protocol-specific message. The **duration** flag is used to specify the persistence of the information. Using the **duration** flag, the user information may be specified to persist as long as the current or next call, or for all calls (including the current call). When the duration is specified to be all calls on the specified line device, the user information is valid and utilized for all calls until the device is eventually closed via **gc_Close( )**.

In the case of **gc_GetUserInfo( )**, the user information is retrieved from an already received protocol-specific message that has been received from the remote side. Note that the user information parameters returned from the call control library in the GC_PARM_BLK buffer must be processed or copied prior to the next Dialogic® Global Call API function call. The reason for this is that the GC_PARM_BLK buffer will be deallocated within Dialogic® Global Call API in a subsequent function call.

Alarm Handling 8

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

- Alarm Handling Overview. 97
- Operation and Configuration of GCAMS 99
- Sample Alarm Scenarios 105

8.1 Alarm Handling Overview

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

There are basically two sources of Dialogic® Global Call API 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 Dialogic® Global Call API call control library that manages alarms is called the Global Call Alarm Management System (GCAMS). GCAMS is initialized automatically when Dialogic® Global Call API is started.

GCAMS provides Dialogic® Global Call API 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.

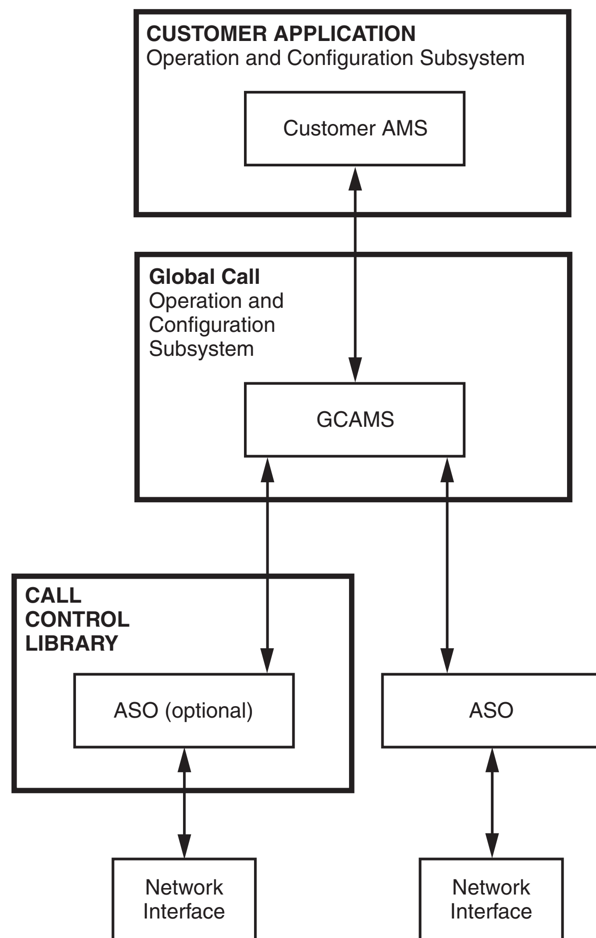
The following sections describe the components and operation of GCAMS.

8.1.1 Alarm Management System Components

The alarm management system (AMS) is made up of several components, including GCAMS. The other components are the customer application’s 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 21 illustrates the relationship between the alarm management system components.

Figure 21. Architectural Diagram of Alarm Management Components



The customer application is responsible for configuring the behavior of GCAMS, including the designation of 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 101.

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 99.

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
- Starting and Stopping Alarm Transmission (E1, T1 and ISDN technologies only)
- Retrieving Alarm Data

8.2.1 Generation of Events for Blocking Alarms

Dialogic® Global Call API 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 101.

The GCEV_BLOCKED and GCEV_UNBLOCKED events are unsolicited events that are sent in addition to other Dialogic® Global Call API 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 73. 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 or trunk level and the channel level:

Logical or Virtual Board (IP technology) or Trunk (E1, T1 or ISDN technology) Level

When the Dialogic® Global Call API recognizes a blocking *alarm on* condition at the logical or virtual board or trunk level, a GCEV_BLOCKED event is generated for the logical or virtual board or trunk 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 or trunk 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 Dialogic® Global Call API recognizes a blocking *alarm off* condition at the logical or virtual board or trunk level, a GCEV_UNBLOCKED event is generated for the logical or virtual board or trunk 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 or trunk 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 Dialogic® 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 Dialogic® 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.

Note: When using Dialogic® Global Call API with Dialogic® Digital Network Interface boards, alarms apply only a the trunk level. An alarm that occurs on a trunk applies to all channels on that trunk.

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 101 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 text 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 8.2.5, "Retrieving Alarm Data", on page 104.

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 Dialogic® 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 *Dialogic® 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, Dialogic® Global Call API 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 *Dialogic® 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:

1. Convert the ASO name to the ASO ID using the **gc_AlarmSourceObjectNameToID( )** function.
2. 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 Starting and Stopping Alarm Transmission

Note: This section applies to E1, T1 and ISDN technologies only.

GCAMS is automatically started when Dialogic® Global Call API is started. However, to begin the transmission of alarms to the remote side, the **gc_TransmitAlarms( )** function must be called. The **gc_TransmitAlarms( )** function sends all alarms as specified in the ALARM_LIST data structure for a given alarm source object.

To stop the transmission of alarms to the remote side, use the **gc_StopTransmitAlarms( )** function.

8.2.5 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)

8.2.5.1 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 its text name, 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 *Dialogic® Global Call API Library Reference* for ASOs that are common to multiple call control libraries.

gc_AlarmNumberToName( )
: converts the alarm number to its text name

8.2.5.2 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 *Dialogic® Global Call API Library Reference*. The names of ASOs that are part of a call control library are provided in the appropriate Global Call 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 *Dialogic® 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
- Scenario 3: Alarm Transmission (E1, T1 and ISDN technologies only)

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

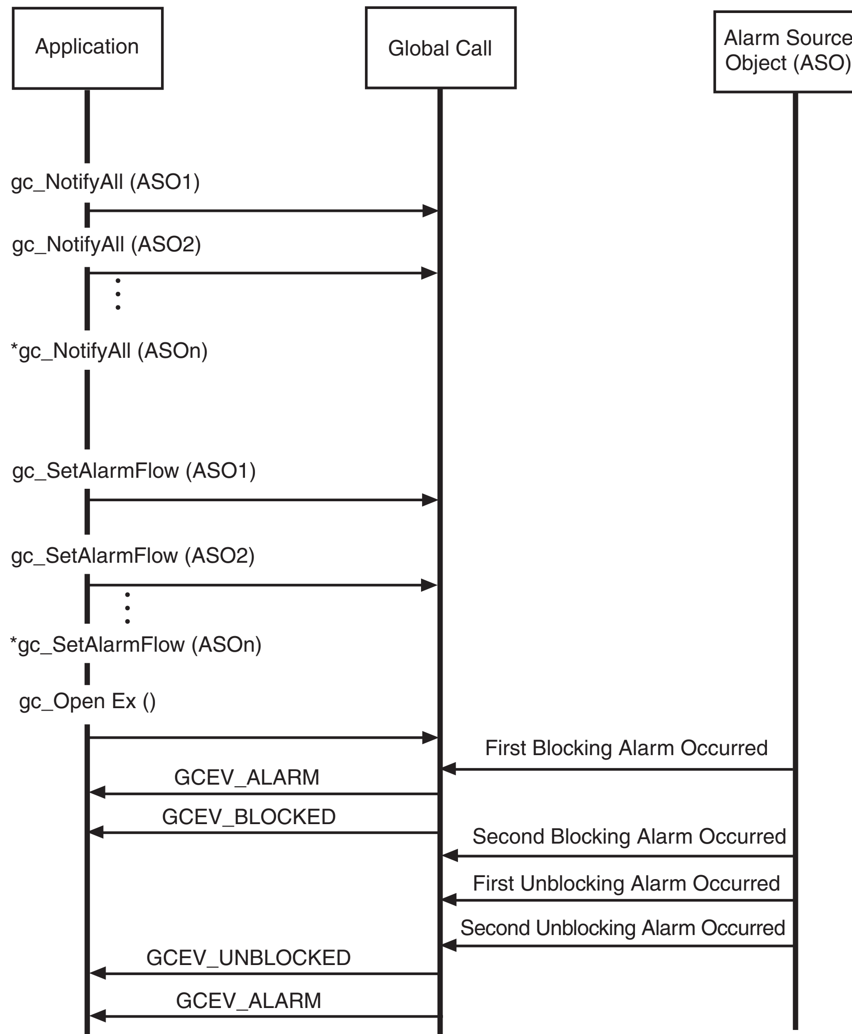
In this scenario, the application wants to be notified of only the first and last blocking alarm events. The default blocking configuration is acceptable. See Figure 22.

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 22. Notification of First and Last Blocking Alarm

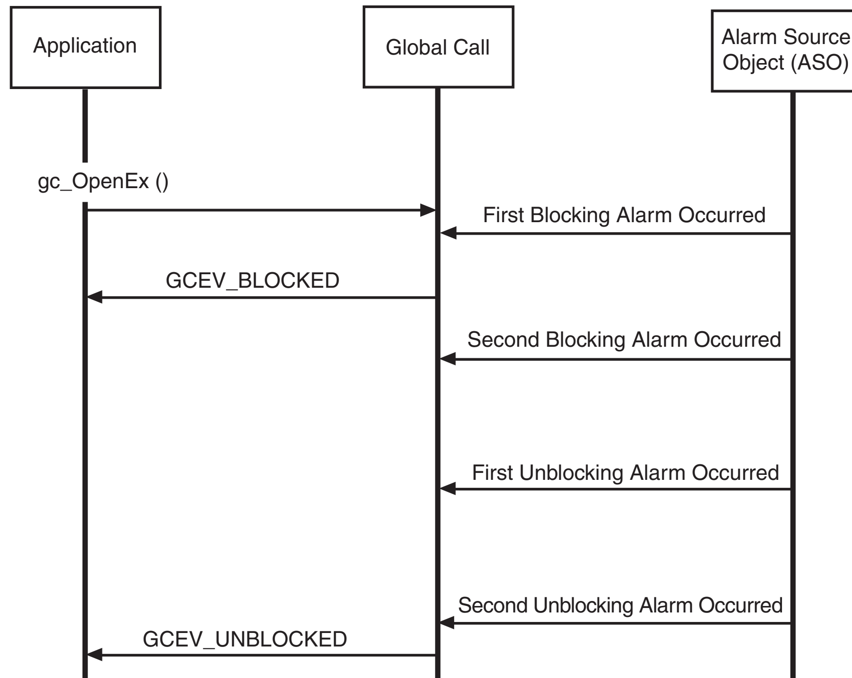


Note: The function calls for alarm processing are not shown.

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 23.

Figure 23. Default Behavior for Alarm Notification

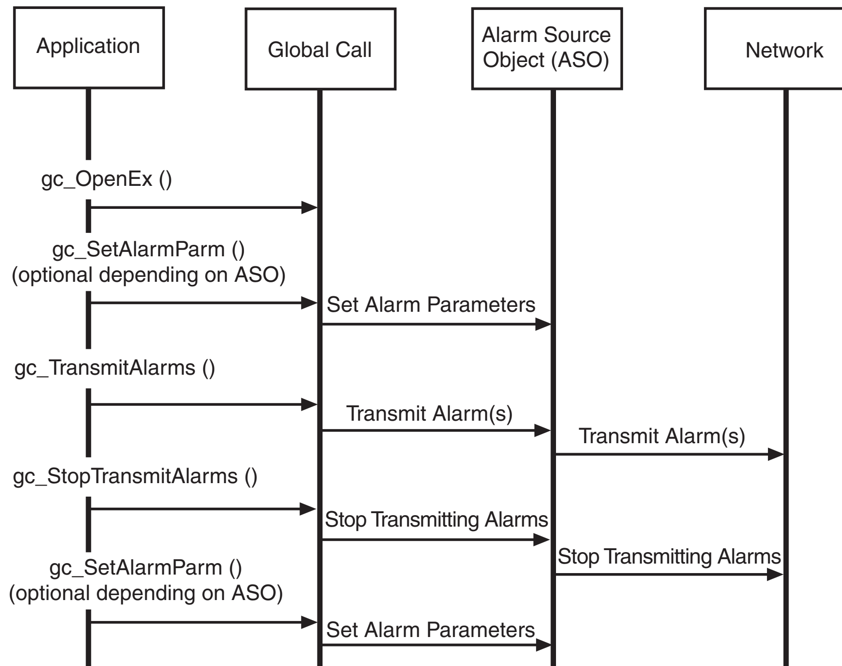


8.3.3 Scenario 3: Alarm Transmission

Note: This scenario applies to E1, T1 and ISDN technologies only.

Figure 24 shows a scenario that demonstrates the sequence of function calls and the actions that they cause in the transmission of alarms.

Figure 24. Alarm Transmission



Real Time Configuration Management 9

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

- Real Time Configuration Manager Overview 109
- RTCM Components 110
- Using RTCM Parameters 112
- Getting and Setting Parameter Information 114
- Handling RTCM Errors 117
- Configuration Procedure 118
- Sample Scenarios Using the RTCM API Functions 119

9.1 Real Time Configuration Manager Overview

The Dialogic® Global Call API Real Time Configuration Management (RTCM) system manages run time configuration for Dialogic® Global Call API components. The RTCM feature is used when the application needs to retrieve (E1, T1 and ISDN technology only) or modify (all technologies) configuration data. If the configuration data is not modified, the application uses the initial values for the configuration.

Note: Not all technologies support the RTCM feature. Refer to the appropriate Dialogic® Global Call API Technology Guide to determine if RTCM is supported.

The Dialogic® Global Call API RTCM system allows applications to:

- Get (E1, T1 or ISDN technologies only) or set (all technologies) the configuration of a protocol dynamically. For example, the default values of country dependent parameters (CDP) can be retrieved or updated with new values.
- Get (E1, T1 or ISDN technologies only) or set (all technologies) 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.
- Get (E1, T1 or ISDN technologies only) or set (all technologies) 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.
- Query (E1, T1 or ISDN technologies only) the protocol ID from the given protocol name or CDP parameter ID from the given CDP parameter name.

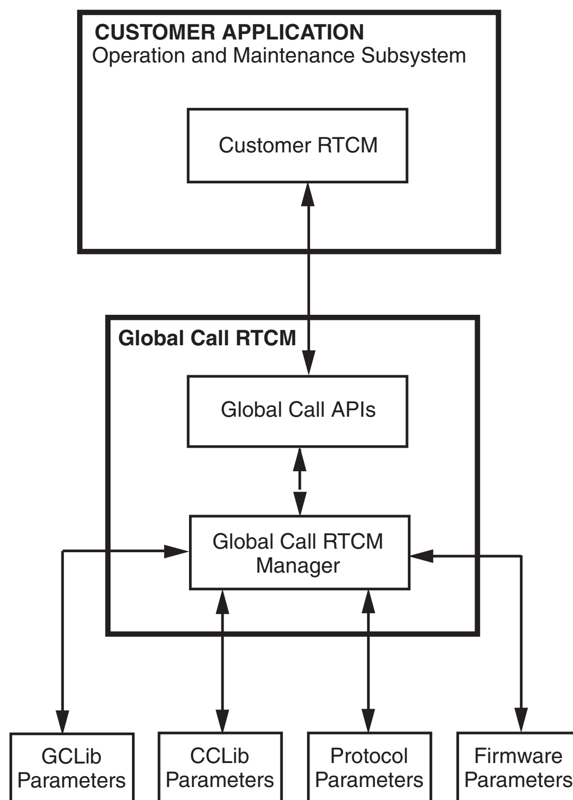
In addition, the RTCM feature provides Dialogic® Global Call API applications with the ability to retrieve configuration parameter information. Some of the ways this information can be used include:

- 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 Dialogic® Global Call API call modules or call events

9.2 RTCM Components

The RTCM comprises three major components: the customer application using RTCM, the Dialogic® Global Call API RTCM, which consists of the Dialogic® Global Call API RTCM APIs and the Dialogic® Global Call API RTCM Manager, and the RTCM parameters. Figure 25 shows the relationship between these components.

Figure 25. Relationship of Customer Application, Dialogic® Global Call API RTCM, and RTCM Parameters



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

9.2.1 Customer Application Using Dialogic® Global Call API RTCM

The customer application interfaces with the Dialogic® Global Call API 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 appropriate Dialogic® Global Call API Technology Guide.
- Ensuring that the configurable parameters match the target entity and inserting parameter data in the proper data format.
- Choosing the proper Dialogic® Global Call API 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 Dialogic® Global Call API RTCM retrieval events.
- Correcting errors in input configurable parameter data based on the Dialogic® Global Call API error messages.

9.2.2 Dialogic® Global Call API RTCM

The Dialogic® Global Call API 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 Dialogic® Global Call API RTCM comprises the RTCM Manager and the RTCM API functions.

The RTCM Manager is responsible for configuring components, including the Dialogic® Global Call API Library (GCLib), Call Control Library (CCLib), protocol, and firmware parameters (see Section 9.3, "Using RTCM Parameters", on page 112).

The RTCM API functions are used to get, set, or query 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 Dialogic® Global Call API RTCM maintains the information about a target object with its associated software module so that the Dialogic® Global Call API RTCM can call the appropriate software module to execute the configuration request. The Dialogic® Global Call API 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: When using IP technology, only the **gc_SetConfigData( )** function is supported. This function is supported in asynchronous mode and can operate on a board device only.

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

Since the Dialogic® Global Call API 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: Dialogic® Global Call API 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 Dialogic® Global Call API RTCM provides a generic way of getting (E1, T1 and ISDN technologies) and setting (all technologies) 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 appropriate Dialogic® Global Call API 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 appropriate Dialogic® Global Call API Technology Guide for more information about configurable parameters.

Parameters in Firmware Module
: parameters that are defined and maintained in a firmware module. See the appropriate Dialogic® Global Call API 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 Dialogic® Global Call API RTCM API function. An example of a target object is (GCTGT_CCLIB_CHAN, Dialogic® Global Call API line device ID).

Set ID and parameter ID
: This uniquely represents a parameter within a specified target object. See Section 9.4, "Getting and Setting Parameter Information", on page 114 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 Dialogic® Global Call API 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 appropriate Dialogic® Global Call API Technology Guide for information about parameter usage.

9.3.2 Parameter Definitions

GCLib or CCLib parameter descriptions can be found in the *Dialogic® Global Call API Library Reference*. Other target objects and their associated set IDs and parameters are described in the appropriate Dialogic® Global Call API Technology Guide. The Dialogic® Global Call API Technology Guides also include 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, “Getting and Setting Parameter Information”, on page 114 and the appropriate Dialogic® Global Call API Technology Guide for detailed information.

9.4 Getting and Setting Parameter Information

The Dialogic® Global Call API RTCM feature supports the retrieval (E1, T1 and ISDN technologies only) or updating (all technologies) of multiple parameters of the same target object in a single Dialogic® Global Call API function call. The functions used to get and set configuration data are as follows:

gc_GetConfigData( ) (E1, T1 and ISDN technologies only)
: retrieves the configuration data from a given target object

gc_SetConfigData( ) (all technologies)
: 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 (all technologies) and the **gc_GetConfigData( )** function (E1, T1 and ISDN technologies only), the configuration data is required to be a generic GC_PARM_BLK data structure. The Dialogic® Global Call API 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 Dialogic® Global Call API utility functions must be used to allocate or deallocate the GC_PARM_BLK memory, insert a parameter, or retrieve a parameter. See the *Dialogic® 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 Dialogic® Global Call API RTCM API functions.

9.4.2 Control Parameters

The Dialogic® Global Call API RTCM API control parameters ensure the efficiency of the retrieve (E1, T1 and ISDN technologies only) or update (IP technology) configuration process and that the application program is not blocked. The application can specify:

- the programming mode
- the timeout interval for completing the retrieval or 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. (This parameter does not apply to the **gc_GetConfigData( )** function.)

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_GetConfigData( )** and **gc_SetConfigData( )** functions operate in the asynchronous and synchronous programming modes:

gc_GetConfigData( ) (E1, T1 and ISDN technologies only)
: **Synchronous Mode:** Upon completion of the function call, the retrieved parameter data is still in the original GC_PARM_BLK data block after the **gc_GetConfigData( )** function returns. The function's return value, GC_SUCCESS, indicates that all requested parameters in a given target object have been successfully retrieved. Other return values indicate that at least one requested parameter in the target object failed to be retrieved due to an error. The **gc_ErrorInfo( )** function is called immediately to obtain the last error and the additional message, which describes the parameter and the error (pointer to the additional message field). During the **gc_GetConfigData( )** function call, once an error occurs, Dialogic® Global Call API stops retrieving the remaining parameters and returns an error value to the application. If this function call is retrieving multiple parameters, the parameters before the error may have been retrieved while other parameters will not have had a chance to be retrieved.

gc_GetConfigData( ) (E1, T1 and ISDN technologies only)
: **Asynchronous Mode:** Upon completion of the function call, the Dialogic® Global Call API application receives the GCEV_GETCONFIGDATA event if all requested parameters have been successfully retrieved. Otherwise, the Dialogic® Global Call API application receives the GCEV_GETCONFIGDATA_FAIL event, which means at least one requested parameter of this request failed to retrieve due to an error. The METAEVENT data structure associated with both events has a field evtdatap that points to a GC_RTCM_EVTDATA data structure. In the GC_RTCM_EVTDATA event, the retrieved _parmblkp field points to the retrieved parameter data. The error value and additional message describing the parameter and the error are also provided in GC_RTCM_EVTDATA data structure.

Note: The **gc_GetConfigData( )** function cannot be called in asynchronous mode for the following target types: GCTGT_GCLIB_SYSTEM, GCTGT_CCLIB_SYSTEM, GCTGT_PROTOCOL_SYSTEM, and GCTGT_FIRMWARE_SYSTEM. The function returns invalid target type. The **gc_GetConfigData( )** function must be called in synchronous mode for these target types.

gc_SetConfigData( ) (all technologies)

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, the Dialogic® Global Call API 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( ) (all technologies)

Asynchronous Mode: The Dialogic® Global Call API application receives the GCEV_SETCONFIGDATA event if all the requested parameters in a given target object are successfully updated. Otherwise, the Dialogic® Global Call API 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.

Note: When using E1, T1 and ISDN technologies, the **gc_SetConfigData( )** function cannot be called in asynchronous mode for the following target types: GCTGT_GCLIB_SYSTEM, GCTGT_CCLIB_SYSTEM, GCTGT_PROTOCOL_SYSTEM, and GCTGT_FIRMWARE_SYSTEM. The function returns invalid target type. The **gc_SetConfigData( )** function must be called in synchronous mode for these target types.

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

9.4.2.2 Timeout Option

When using IP technology, the timeout option provided by the timeout parameter in the **gc_SetConfigData( )** function is not supported and should be set to 0.

When using E1, T1 and ISDN technology, the following apply:

- The customer application can specify the timeout for completing the parameter retrieval or update. The **gc_GetConfigData( )** and **gc_SetConfigData( )** functions support the timeout option only in synchronous mode. When a timeout occurs in the synchronous mode, the function returns an EGC_TIMEOUT error to the application. The timeout option is ignored if the function is executed in asynchronous mode.
- The function call is stopped immediately when a timeout occurs. When accessing multiple parameters in a single function call, some, but not all, parameters may have been retrieved or updated before the timeout.
- A timeout value selected to be less than or equal to zero indicates an infinite timeout. When the **gc_SetConfigData( )** function has an infinite timeout set and is updated at the Null call state, this thread is blocked if the target object still has any active call. The customer application can avoid this situation by using the asynchronous mode or multi-threading technology.

9.4.2.3 Update Condition

When using the **gc_SetConfigData( )** function 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 calls, 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 13 describes the possible settings and resulting actions for the update condition as used by the **gc_SetConfigData( )** function.

Table 13. Update Condition Flag and Dialogic® Global Call API Process

Update condition flag (Dialogic® Global Call API APP)	Parameter Update Allowed in Target Object	Target Object Status	Dialogic® Global Call API 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 Dialogic® Global Call API 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 *Dialogic® Global Call API Library Reference* for Dialogic® Global Call API RTCM error values and messages.

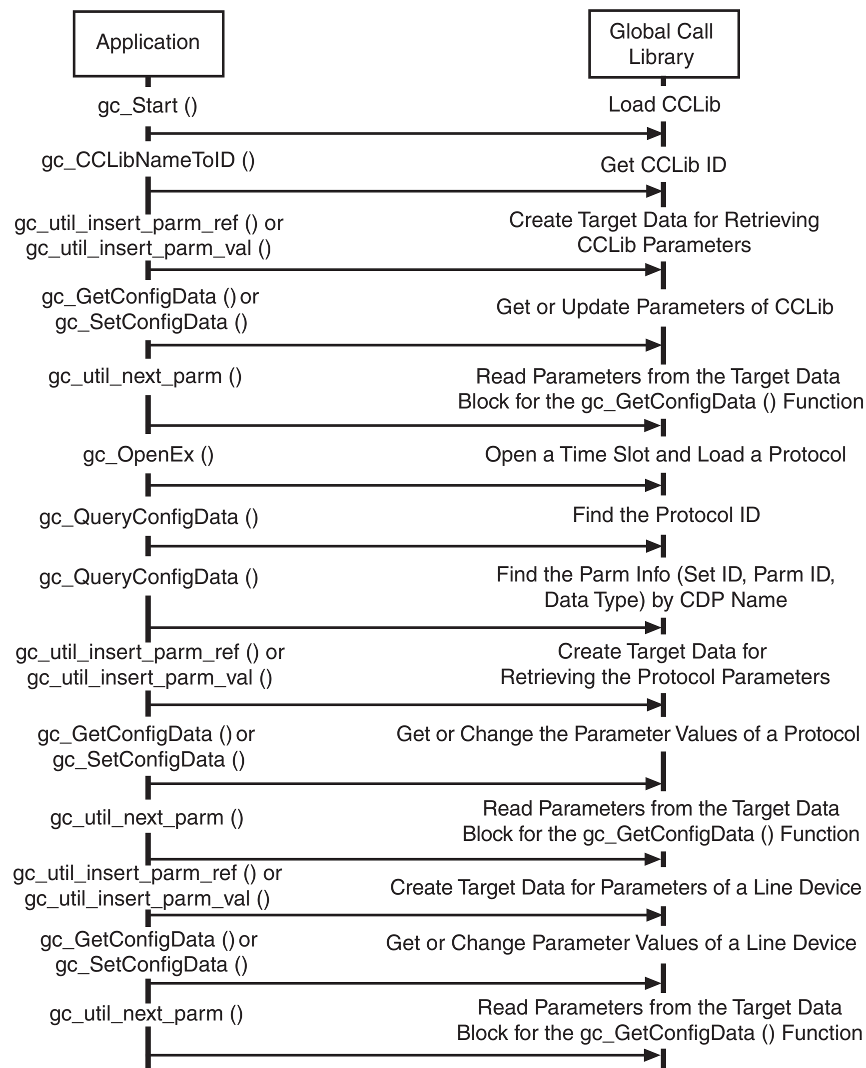
9.6 Configuration Procedure

The basic steps for using the Dialogic® Global Call API 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 Dialogic® Global Call API to finish the request efficiently without blocking the application program.

Figure 26 illustrates the run-time configuration procedure.

Figure 26. Run Time Configuration Procedure



Note: In Figure 26, for IP technology, only the **gc_SetConfigData( )** function is supported for changing the parameter values of a board device. See Section 9.7.3, “Getting or Setting Line Device Configuration in Synchronous Mode”, on page 122.

9.7 Sample Scenarios Using the RTCM API Functions

This section shows the following examples in which the customer application uses the Dialogic® Global Call API RTCM API functions to get or set the configuration of various target objects. With the exception of the last example, all of these apply to E1, T1 and ISDN technologies only. The examples include:

- Getting or Setting GCLib Configuration in Synchronous Mode

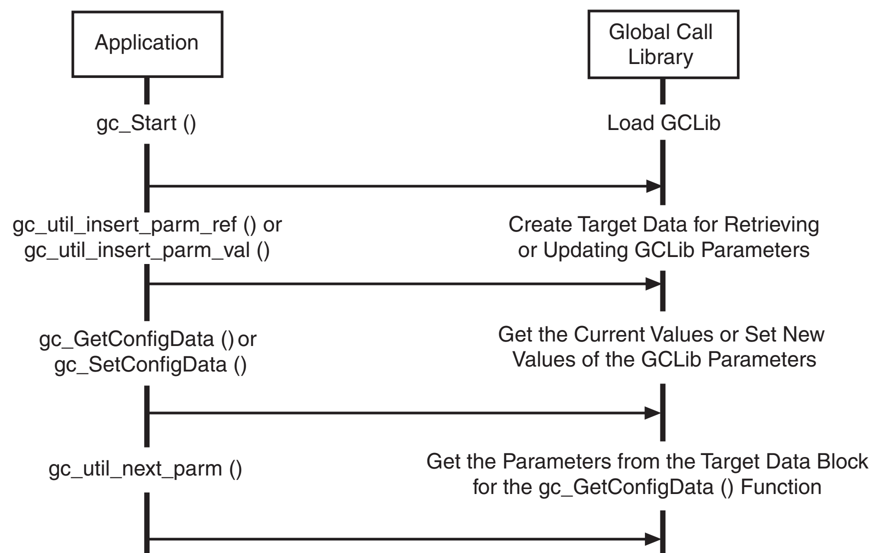
- Getting or Setting CCLib Configuration in Synchronous Mode
- Getting or Setting Line Device Configuration in Synchronous Mode
- Setting Line Device Configuration in Asynchronous Mode
- Setting Board Device Configuration in Asynchronous Mode (IP Technology)

9.7.1 Getting or Setting GCLib Configuration in Synchronous Mode

Note: This section applies to E1, T1 and ISDN technologies only.

The Dialogic® Global Call API RTCM feature allows the customer application to retrieve or change the default configuration of a GCLib even before any line device is opened. Figure 27 shows the procedure for synchronous mode.

Figure 27. Getting or Setting GCLib Configuration in Synchronous Mode



The following describes the procedure for getting or setting the configuration of the GCLib in synchronous mode:

1. Load the GCLib (after the **gc_Start( )** function is called).
2. Create the target object data (a GC_PARM_BLK data structure) with the appropriate set ID, parm ID, value size, and value, if applicable, by calling the Dialogic® Global Call API utility functions **gc_util_insert_parm_ref( )** or **gc_util_insert_parm_val( )**. See the *Global Call API Library Reference* for more information.
3. Call the **gc_GetConfigData( )** or **gc_SetConfigData( )** function with: **target_type** = GCTGT_GCLIB_SYSTEM
   target_id = 0
   time_out = 0
   mode = EV_SYNC

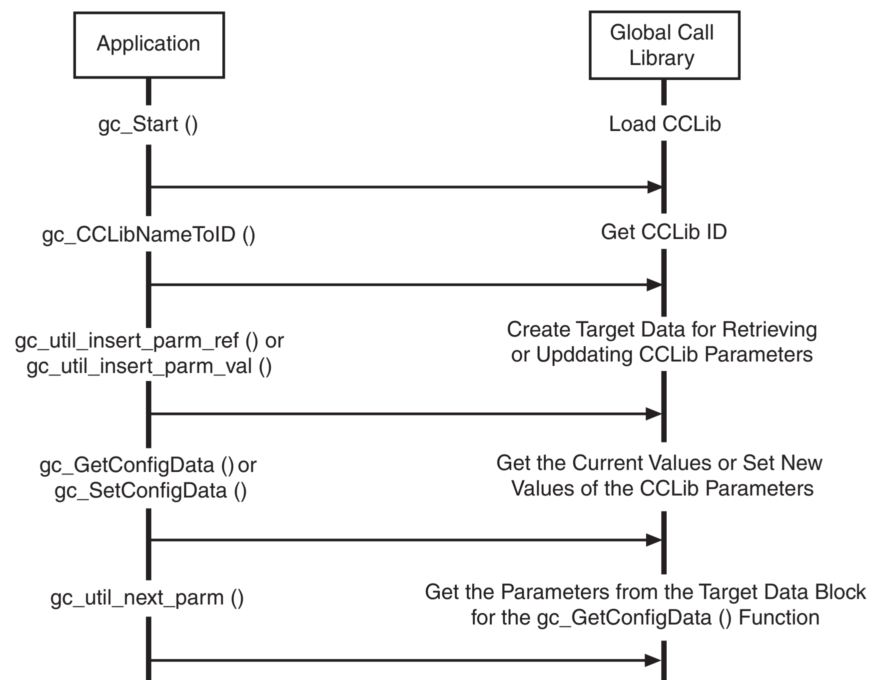
4. If the **gc_GetConfigData( )** function returns successfully, then obtain the individual parameter data by calling the **gc_util_get_next_parm( )** function. If an error occurs, call the **gc_ErrorInfo( )** function to find the error and correct it.

9.7.2 Getting or Setting CCLib Configuration in Synchronous Mode

Note: This section applies to E1, T1 and ISDN technologies only.

The Dialogic® Global Call API RTCM feature allows the customer application to retrieve or change the default configuration of a CCLib even before any line device is opened. Figure 28 shows the procedure for synchronous mode.

Figure 28. Getting or Setting CCLib Configuration in Synchronous Mode



The following describes the procedure for getting or setting the configuration of a CCLib in synchronous mode.

1. Load the call control library after the **gc_Start( )** function is called.
2. Find the CCLib ID using its name by calling the **gc_CCLibNameToID( )** function. If the application has doubt about the CCLib name, it can call the **gc_GetCCLibStatusAll( )** function to verify whether the CCLib has been started.
3. Create the target object data (a GC_PARM_BLK data structure) with the appropriate set ID, parm ID, value size, and value, if applicable, by calling the Dialogic® Global Call API utility functions **gc_util_insert_parm_ref( )** or **gc_util_insert_parm_val( )**. See the *Dialogic® Global Call API Library Reference* for more information.

4. Call the **gc_GetConfigData( )** or **gc_SetConfigData( )** function with:
   target_type = GCTGT_CCLIB_SYSTEM
   target_id = CCLib ID
   time_out = 0
   mode = EV_SYNC
5. If the **gc_GetConfigData( )** function returns successfully, then obtain the individual parameter data by calling the **gc_util_get_next_parm( )** function. If an error occurs, call the **gc_ErrorInfo( )** function to find the error and correct it.

9.7.3 Getting or Setting Line Device Configuration in Synchronous Mode

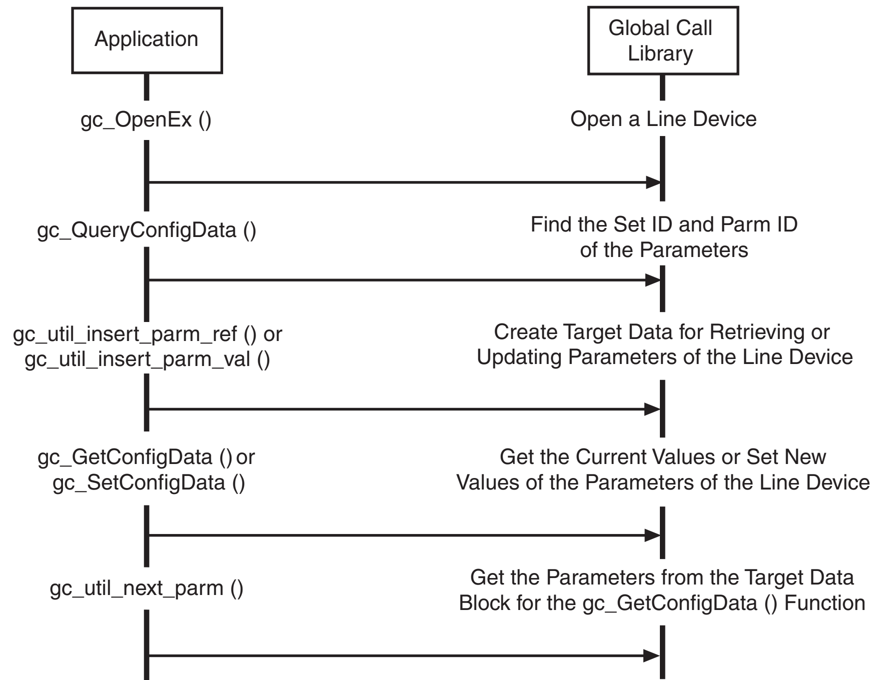
Note: This section applies to E1, T1 and ISDN technologies only.

The Dialogic® Global Call API RTCM feature also allows the customer application to retrieve or change the default configuration of a line device in synchronous mode. Synchronous mode can be used effectively in any of the following cases:

- The request is to retrieve parameters.
- The request is to update parameters that are NOT call related.
- The request is to update parameters that are call related but there is no active call on the target object.
- The **target type** is neither GCTGT_FIRMWARE_CHAN nor GCTGT_FIRMWARE_NETIF (that is, the parameters are not maintained in the firmware).

Figure 29 shows the procedure for getting or setting line device configuration in synchronous mode.

Figure 29. Getting or Setting Line Device Configuration in Synchronous Mode



The following describes the procedure for getting or setting the configuration of a Line Device:

1. Open the line device (by calling the **gc_OpenEx( )** function) and get the line device ID.
2. If the parameters of the line device are protocol CDP parameters, use an approach similar to getting the CDP **parameter ID** described in the "Getting or Setting Protocol Configuration in Synchronous Mode" section.
3. Create the target object data (a GC_PARM_BLK data structure) with the appropriate set ID, parm ID, value size, and value, if applicable, by calling the Dialogic® Global Call API utility functions. See the *Dialogic® Global Call API Library Reference* for more information on the utility functions.
4. Call the **gc_GetConfigData( )** or **gc_SetConfigData( )** function with:
   target_type = GCTGT_CCLIB_NETIF,GCTGT_PROTOCOL_NETIF, GCTGT_CCLIB_CHAN or GCTGT_PROTCOL_CHAN
   target_id = Dialogic® Global Call API line device ID
   time_out > 0
   mode = EV_SYNC
   update condition = GCUPATE_IMMEDIATE (**gc_SetConfigData( )** function only)
5. If the **gc_GetConfigData( )** function returns successfully, obtain the individual parameter data by calling the **gc_util_get_next_parm( )** function. If an error occurs, call the **gc_ErrorInfo( )** function to find the error and then correct it.

9.7.4 Setting Line Device Configuration in Asynchronous Mode

Note: This section applies to E1, T1 and ISDN technologies only.

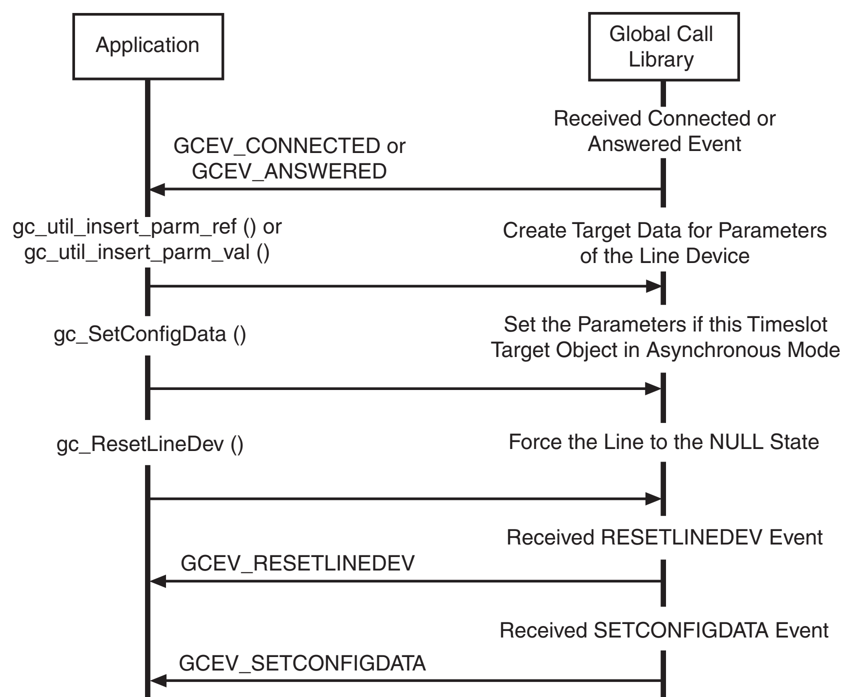
The Dialogic® Global Call API RTCM allows the customer application to retrieve or change the default configuration of a line device in asynchronous mode.

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_FIRMWARE_CHAN or GCTGT_FIRMWARE_NETIF (that is, the parameters are maintained in firmware).

Figure 30 shows the procedure for setting line device configuration in asynchronous mode.

Figure 30. Setting Line Device Configuration in Asynchronous Mode (E1, T1 and ISDN Technology)



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 Dialogic® Global Call API utility functions. See the *Dialogic® Global Call API Library Reference* for more information.
2. Call the **gc_SetConfigData**( ) function with:
target_type = GCTGT_CCLIB_NETIF, GCTGT_PROTOCOL_NETIF,

GCTGT_FIRMWARE_NETIF, GCTGT_CCLIB_CHAN, GCTGT_PROTCOL_CHAN, or GCTGT_FIRMWARE_CHAN
target_id = Dialogic® Global Call API line device ID
time_out = 0
mode = EV_ASYNC
update condition = GCUPATE_ATNULL

3. Call the **gc_ResetLineDev( )** function to enforce the line to the NULL state.
4. If the **gc_ResetLineDev( )** function is successful, a GCEV_RESETLINE event is received. 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.

9.7.5 Setting Board Device Configuration in Asynchronous Mode (IP Technology)

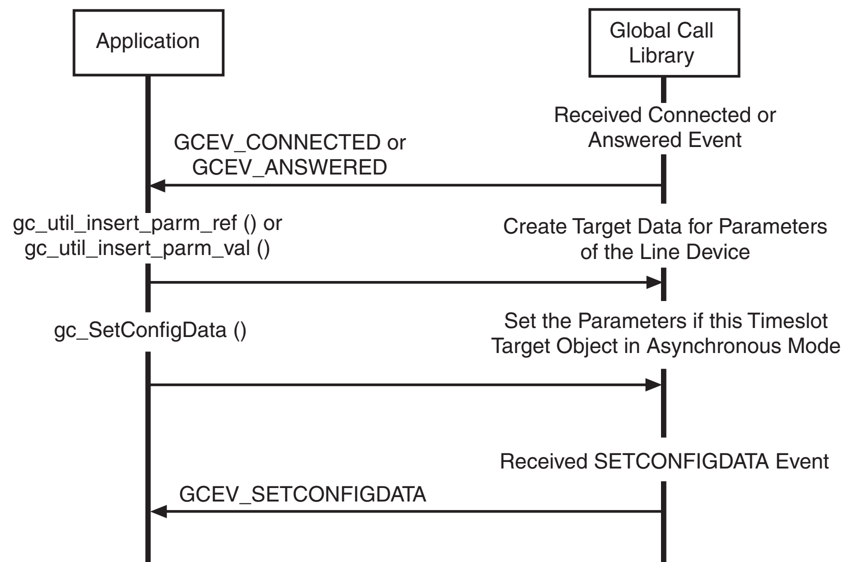
The Dialogic® Global Call API RTCM allows the customer application to change the default configuration of a board device in asynchronous mode.

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 31 shows the procedure for setting board device configuration in asynchronous mode for IP technology.

Figure 31. Setting Board Device Configuration in Asynchronous Mode (IP Technology)



The procedure for setting the configuration of a board device in asynchronous mode for IP technology 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 Dialogic® Global Call API utility functions. See the *Dialogic® Global Call API Library Reference* for more information.
2. Call the **gc_SetConfigData( )** function with:
target_type = GCTGT_CCLIB_NETIF
target_id = Dialogic® Global Call API 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 Dialogic® Global Call API Service Request (GCSR) feature. Topics include the following:

- Service Request Overview 127
- Service Request Components 128
- Service Request Data 129
- General Service Request Scenario 130

10.1 Service Request Overview

The Dialogic® Global Call API 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 Dialogic® Global Call API application needs to make a request between two Dialogic® Global Call API 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 in a given technology are largely dependent on the support provided by the call control libraries for that technology. Refer to the appropriate Dialogic® Global Call API Technology Guide for more information.

Figure 32 shows the architecture of the GCSR feature.

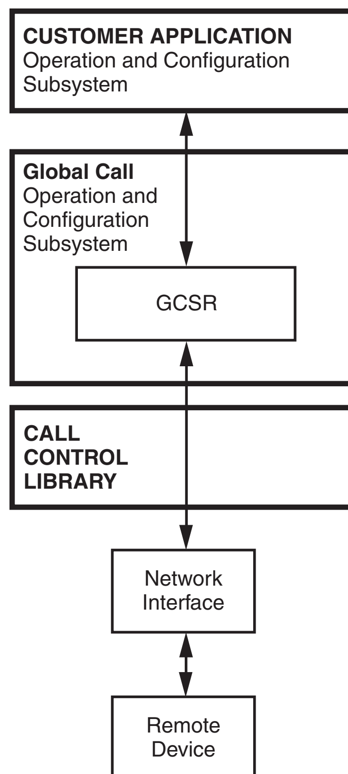


Figure 32. Service Request Architecture

10.2 Service Request Components

Using the Dialogic® Global Call API 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 appropriate Dialogic® Global Call API 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 appropriate Dialogic® Global Call API 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.

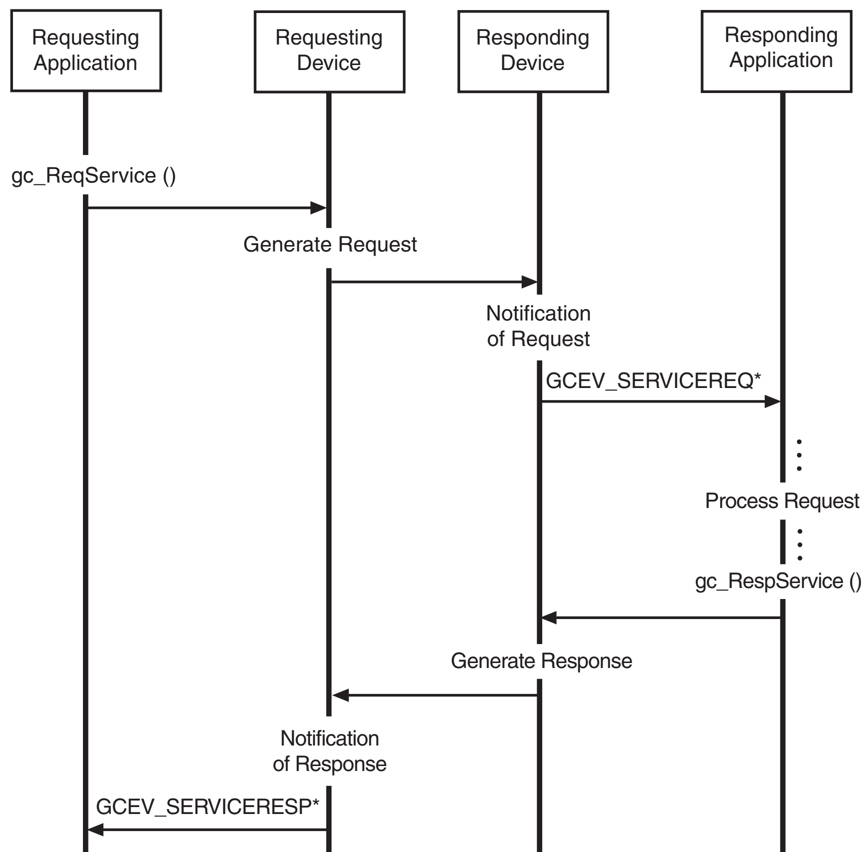
Depending on the call control library used, additional parameters may also be used. Refer to the *Dialogic® Global Call API Library Reference* for more information about the GC_PARM_BLK data structure. Before the Service Request feature can be used, a GC_PARM_BLK data structure must be set up to handle 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 *Dialogic® 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 33 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 appropriate Dialogic® Global Call API Technology Guide for more information.

Figure 33. 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 Dialogic® Global Call API to Implement Call Transfer 11

The information in this chapter is technology independent, however it describes a method of call transfer that is **supported by IP technology only**. For more specific information about implementing call transfer on IP technology, see the *Dialogic® Global Call IP Technology Guide*. For more specific information about implementing call transfer on E1, T1 and ISDN technologies, see Section 3.5, "Advanced Call Control with Call Hold and Transfer", on page 64 and the appropriate Dialogic® Global Call Technology Guide. The topics discussed in this chapter are:

- Introduction to Call Transfer 131
- Call Transfer State Machine 132

11.1 Introduction to Call Transfer

The Dialogic® Global Call API supports the following call transfer methods:

- Blind Call Transfer
- Supervised Call Transfer

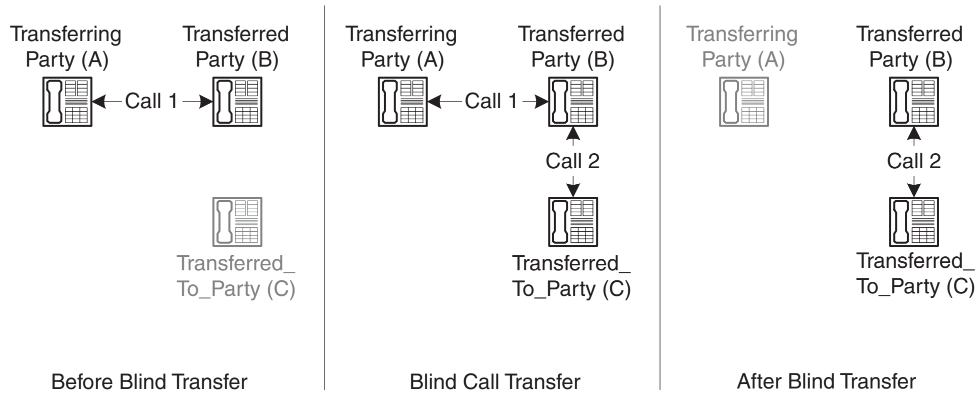
11.1.1 Blind Call Transfer

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 34, party B initiates the transferred call to party C.

Figure 34. Blind Call Transfer (Unsupervised Transfer)

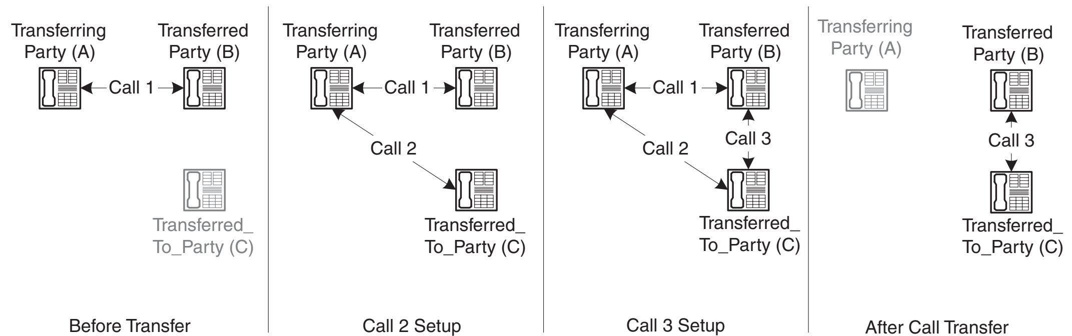


11.1.2 Supervised Call Transfer

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 35. Supervised Call Transfer



11.2 Call Transfer State Machine

Table 14 lists the new Dialogic® Global Call API call states for blind call transfer.

Table 14. New Dialogic® Global Call API 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 36 and Figure 37 apply to the case where party B initiates the transferred call to party C (see Figure 34), and **not** to the case where Party A places the primary call with party B on hold and then calls party C.

Figure 36. Call State Model for Blind Call Transfer at Party A

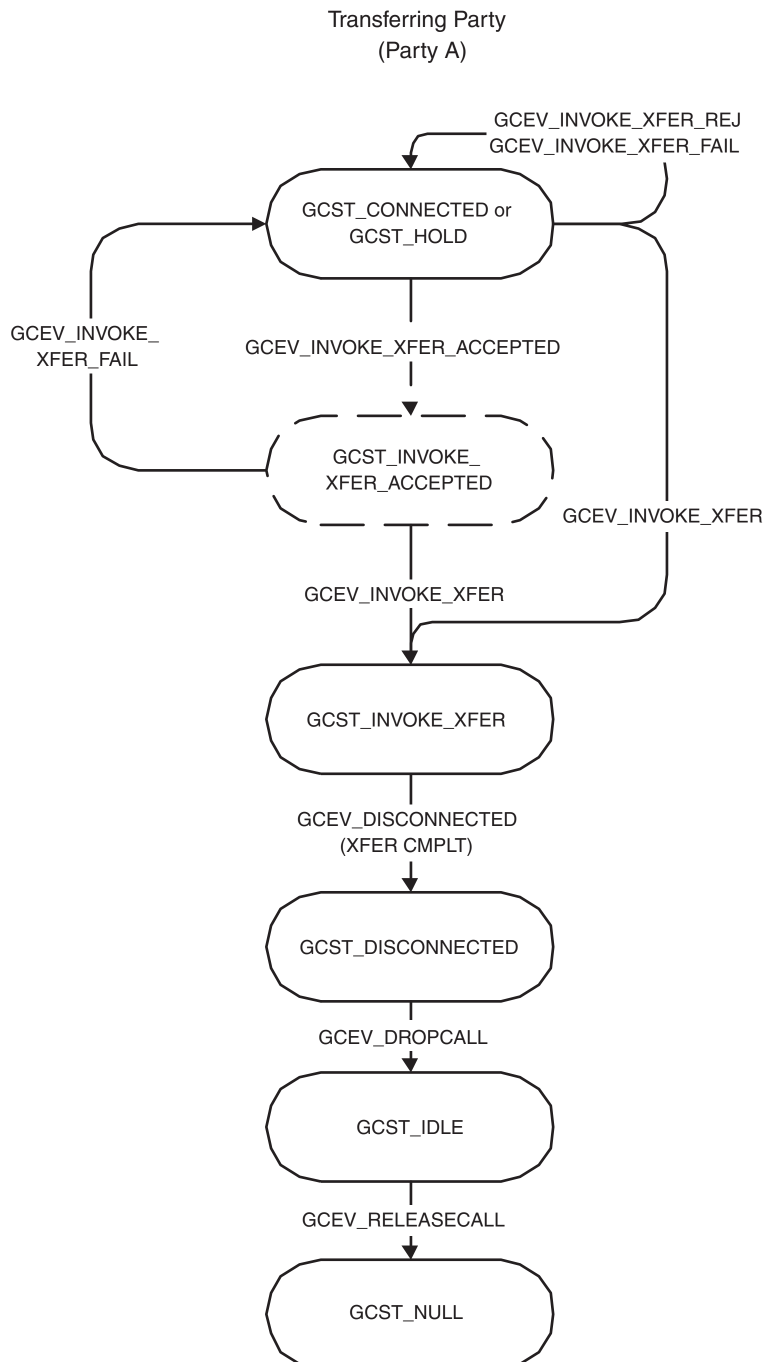
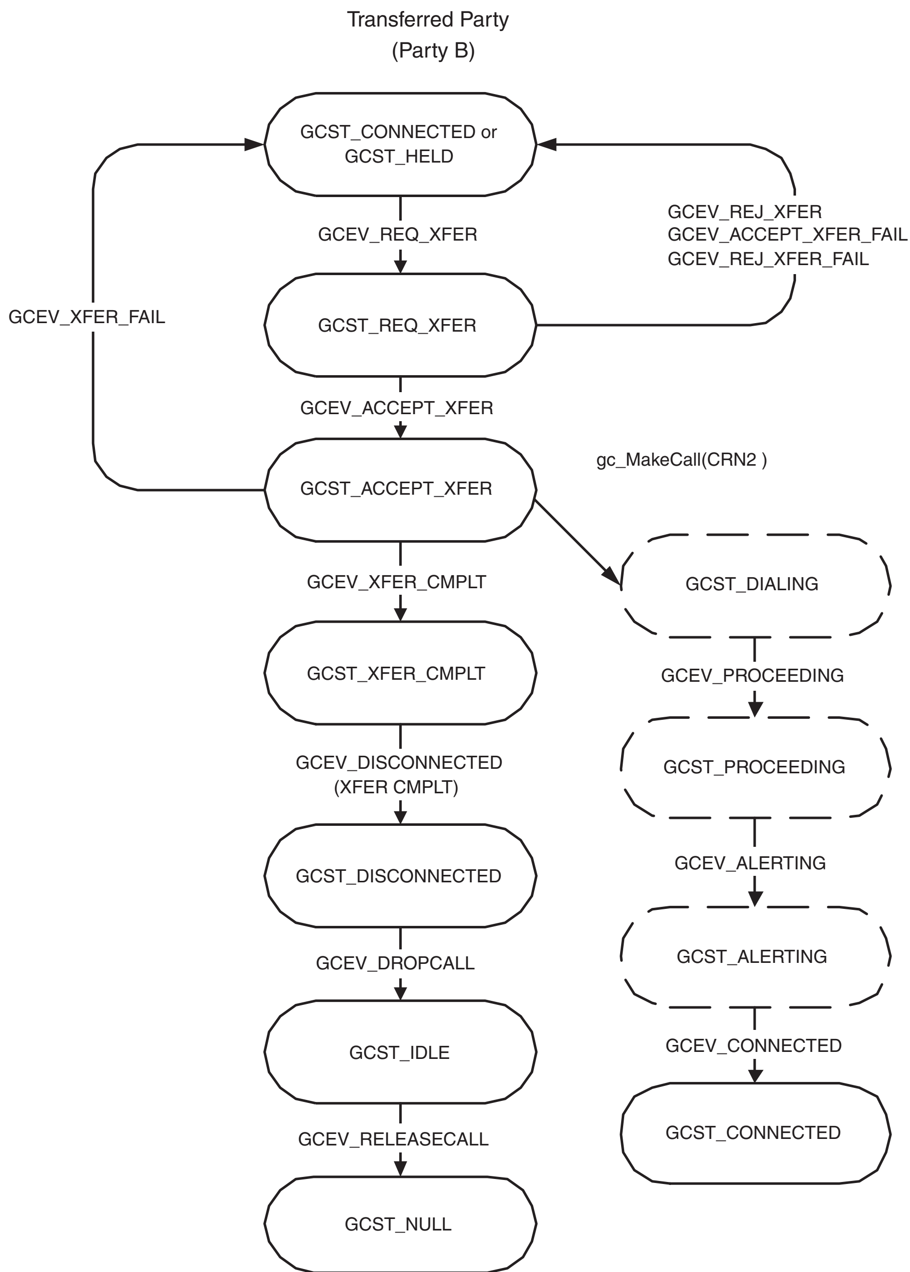


Figure 37. Call State Model for Blind Transfer at Party B



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 38, Figure 39 and Figure 40 apply to the supervised transfer case represented in Figure 35.

Figure 38. Call State Model for Supervised Transfer at Party A

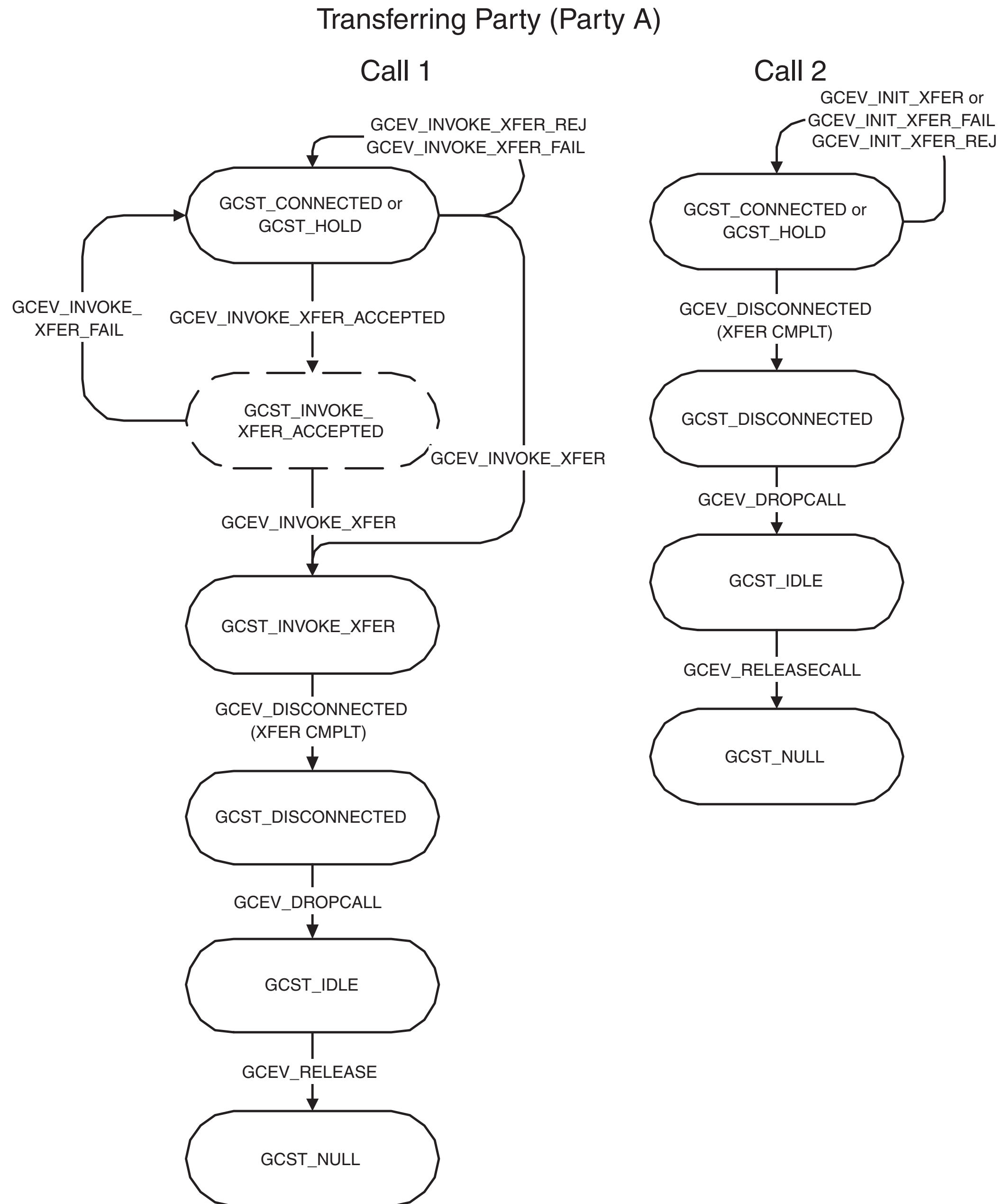


Figure 39. Call State Model for Supervised Transfer at Party B

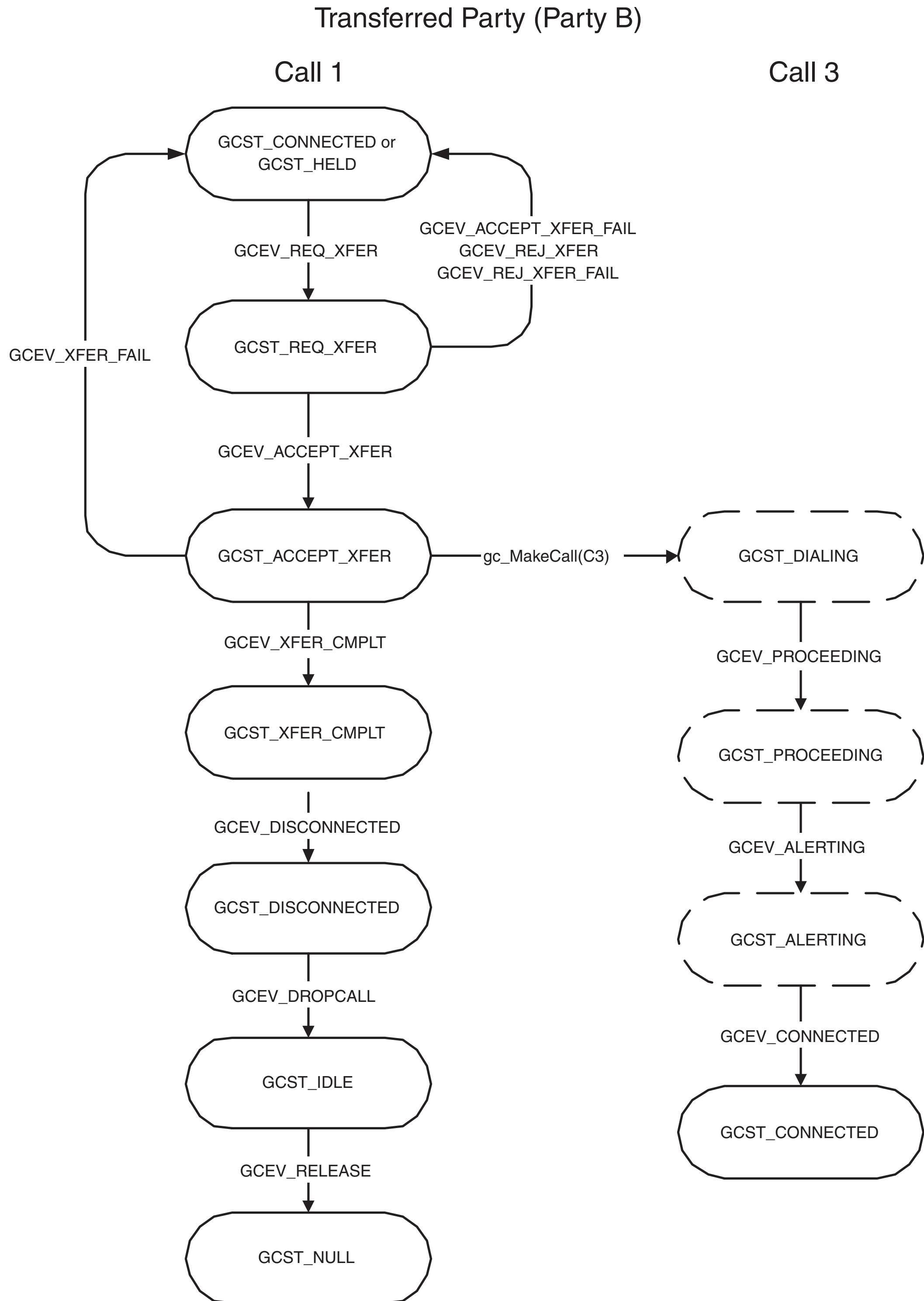
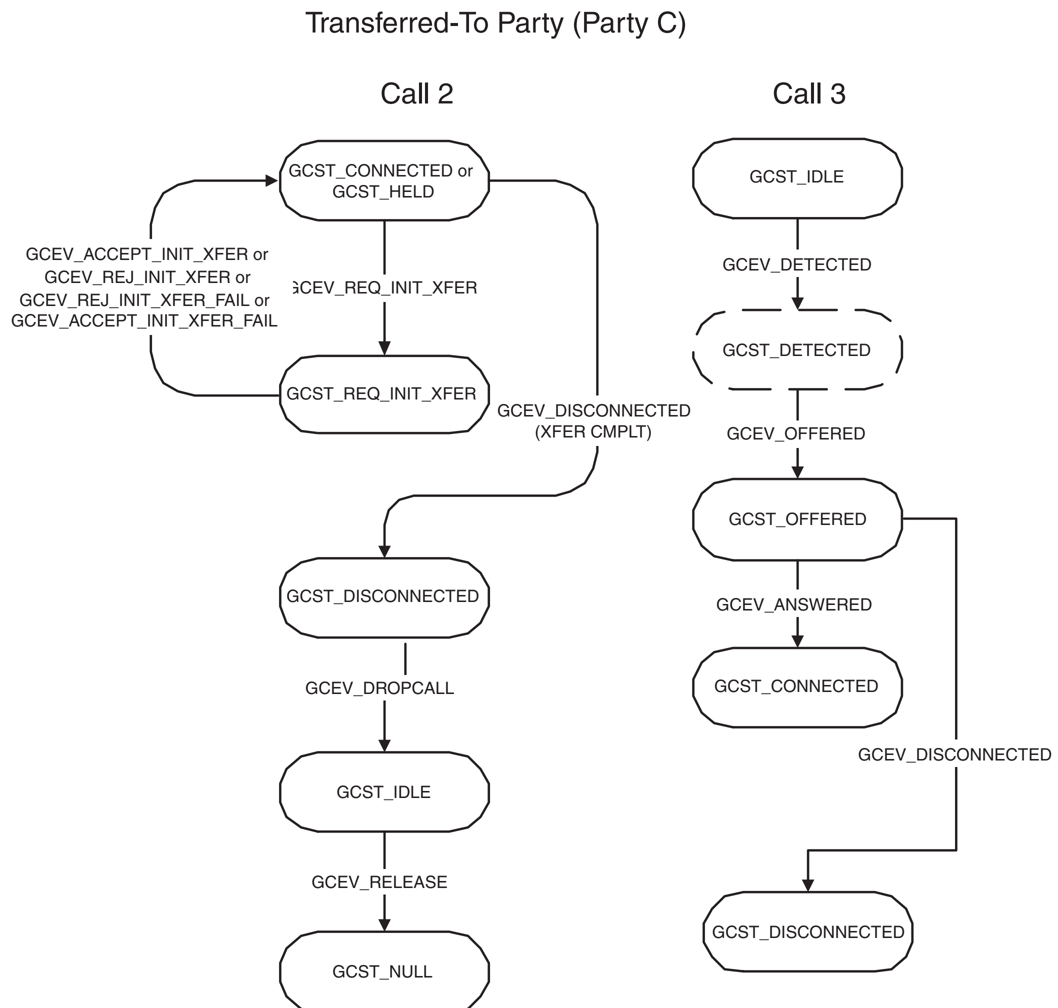


Figure 40. Call State Model for Supervised Transfer at Party C



Building Applications 12

This chapter provides general information for building applications that use the Dialogic® Global Call API software. For additional technology-specific information, refer to the appropriate Dialogic® Global Call API Technology Guide. Topics included in this chapter are:

- Compiling and Linking in Linux 139
- Compiling and Linking in Windows® 140

12.1 Compiling and Linking in Linux

An application that uses the Dialogic® Global Call API software must include references to the Dialogic® Global Call API header files and must include the appropriate library files. This information is provided the following topics:

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

12.1.1 Include Files

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

gclib.h
: primary Dialogic® Global Call API header file

gcerr.h
: header file containing equates for error codes

Note: See the appropriate Dialogic® Global Call API Technology Guide for technology-specific header files.

12.1.2 Required Libraries

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

libgc.so
: the primary Dialogic® Global Call API shared object file. Linking this file is mandatory. Use the **-lgc** argument to the system linker.

libdxxx.so
: the primary Voice shared object file. This library is only required if the application uses R4 voice library functions directly, for example, **dx_play( )**.

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

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: Developers should consider using these variables when compiling and linking applications. The names of the variables will remain constant, but the values may change over time.

12.2 Compiling and Linking in Windows®

An application that uses the Dialogic® Global Call API software must include references to the Dialogic® Global Call API 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

12.2.1 Include Files

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

gclib.h
: primary Dialogic® Global Call API header file

gcerr.h
: header file containing equates for error codes

Note: See the appropriate Dialogic® Global Call API Technology Guide for technology-specific header files.

12.2.2 Required Libraries

The following library files must be linked to the application:

libgc.lib
the primary Dialogic® Global Call API 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( )**.

12.2.3 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: Developers should consider using these variables when compiling and linking applications. The names of the variables will remain constant, but the values may change over time.

12.2.4 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:

libdm3cc.dll (for E1, T1 and ISDN technologies)
DM3 call control library

libgcs7.dll
SS7 call control library

libgch3r.dll
IP call control library

libgcipm.dll
IP call control library

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

Debugging 13

This chapter provides references to other documents that provide detailed information for debugging applications that use the Dialogic® Global Call API.

For general Dialogic® Global Call API debugging information, see the "Runtime Trace Facility (RTF) Reference" chapter in the Dialogic® Host Media Processing Diagnostics Guide.

For debugging information that is technology- or protocol-specific, see the following:

- *Dialogic® Global Call E1/T1 CAS/R2 Technology Guide*
- *Dialogic® Global Call ISDN Technology Guide*
- *Dialogic® Global Call IP Technology Guide*

Glossary

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

ANI-on-Demand: A feature of AT&T ISDN service whereby the user can automatically request caller ID from the network even when caller ID does not exist.

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 Dialogic® Global Call API and successfully started by the Dialogic® Global Call API gc_Start( ) function.

B channel: A bearer channel used in ISDN interfaces. This circuit-switched, digital channel can carry voice or data at 64,000 bits/second in either direction

BC: See *bearer capability.*

bearer capability: A field in an ISDN call setup message that specifies the speed at which data can be transmitted over an ISDN line.

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 Dialogic® Global Call API commands on that line device. Call related functions may not be called with the exception of gc_DropCall( ), gc_ReleaseCall( ) (E1, T1 and ISDN technologies only) 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 Dialogic® Global Call API commands are available to the application.

call analysis: When using Dialogic® Digital Network Interface boards, a term that describes the activity that occurs after a call is connected (post-connect), such as voice detection and answering machine detection. Compare to *call progress*.

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 Dialogic® Global Call API functions to implement network specific commands and communications.

call progress: When using Dialogic® Digital Network Interface boards, a term that describes the activity that occurs before a call is connected (pre-connect), such as busy or ringback. Compare to *call analysis*.

call progress analysis: When using Dialogic® Digital Network Interface boards, a collective term for call progress and call analysis. See also *call progress* and *call analysis*.

call progress tone: When using E1, T1 and ISDN technologies, a tone sent from the PTT to tell the calling party the progress of the call, (for example, a dial tone, busy tone, or ringback tone). The PTT's can provide additional tones, such as a confirmation tone, splash tone or a reminder tone, to indicate a feature in use.

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

call states: Call processing stages in the application.

CAS: Channel Associated Signaling. Signaling protocols in which the signaling bits for each time slot are in a fixed location with respect to the framing. In E1 systems, time slot 16 is dedicated to signaling for all 30 voice channels (time slots). The time slot the signaling corresponds to is determined by the frame number within the multiframe and whether it's the high or low nibble of time slot 16. In T1 systems, the signaling is also referred to as robbed-bit signaling, where the least significant bit of each time slot is used for the signaling bits during specific frames.

CEPT: Conference des Administrations Europeenes des Postes et Telecommunications. A collection of groups that set European telecommunications standards.

compelled signaling: Transmission of next signal is held until acknowledgement of the receipt of the previous signal is received at the transmitting end.

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

congestion: Flow of user-to-user data

CDP: Country Dependent Parameter; see the Dialogic® Global Call API Country Dependent (CDP) Reference for details.

CRN: See Call Reference Number.

CRV: Call Reference Value

D channel: The data channel in an ISDN interface that carries control signals and customer call data in packets. This information is used to control transmission of data on associated B channels.

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: A Dialogic® data path that processes one incoming or outgoing call at a time. Compare to time slot.

digital channel: Designates a bi-directional transfer of data for a single time slot of a T1 or an E1 digital frame between a T1/E1 device that connects to the digital service and the SCbus. Digitized information from the T1/E1 device is sent to the SCbus over the digital transmit channel. The response to this call is sent from the SCbus to the T1/E1 device over the digital receive (listen) channel.

DNIS: Dialed Number Identification Service. A feature of 800 lines that allows a system with multiple 800 lines in its queue to access the 800 number the caller dialed. Also provides caller party number information.

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

Drop and Insert: 1. A process where the information carried by a transmission system is demodulated (dropped) at an intermediate point and different information is entered (inserted) for subsequent transmission. 2. A configuration in which two network interface resources are connected via an internal bus, such as the SCbus, to connect calls from one network interface to the other. A call from one network interface can be dropped to a resource, such as a voice resource, for processing. In return, the resource can insert signaling and audio and retransmit this new bit stream via the internal bus and connect the call to a different channel. Drop and insert configurations provide the ability to access an operator or another call.

E1 CAS: E1 line using Channel Associated Signaling. In CAS, one of the 32 channels (time slot 16) is dedicated to signaling for all of the 30 voice channels.

E1: Another name given to the CEPT digital telephony format devised by the CCITT that carries data at the rate of 2.048 Mbps (DS-1 level).

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 Dialogic® Global Call API and which did not successfully start when the Dialogic® Global Call API 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.

Dialogic® Global Call API: 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.).

Digital Network Interface boards: Dialogic® Digital Network Interface boards that provide E1 and T1 interfaces. These boards provide the physical interfaces for Dialogic® HMP Software applications that require network connectivity.

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

IE: See *Information Element*.

Information Element (IE): Used by the ISDN (Integrated Services Digital Network) protocol to transfer information. Each IE transfers information in a standard format defined by CCITT standard Q.931.

Integrated Services Digital Network: See *ISDN*.

ISDN: Integrated Services Digital Network. An internationally accepted standard for voice, data, and signaling that provides users with integrated services using digital encoding at the user-network interface. Also the name of a call control library configured for the Dialogic® Global Call API.

LAPB: Link Access Protocol Balanced.

LAPD: Link Access Protocol on the D channel.

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

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.

NCAS: Non-Call Associated Signaling. Allows users to communicate by user-to-user signaling without setting up a circuit-switched connection (this signal does not occupy B channel bandwidth). A temporary signaling connection is established and cleared in a manner similar to the control of a circuit-switch connection. Since NCAS

calls are not associated with any B channel, applications receive and transmit NCAS calls on the D channel line device. Once the NCAS connection is established, the application can transmit user-to-user messages using the CRN associated with the NCAS call.

Network Facility Associated Signal: See *NFAS*.

NFAS: Network Facility Associated Signaling; Allows multiple spans to be controlled by a single D channel subaddressing.

Non-Call Associated Signal: See *NCAS*.

NSI: Network Specific Information message.

NT1: Network Terminator. The connector at either end of an ISDN link that converts the two-wire ISDN circuit interface to four wires.

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

overlap viewing: A condition of waiting for additional information about the called party number (destination number).

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. Linux and Windows both use preemptive multitasking to support multiple simultaneous processes.

PRI: Primary Rate Interface. An interface at the ends of high-volume trunks linking CO facilities and ISDN network switches to each other. A T1 ISDN PRI transmits 23 B channels (voice/data channels) and one D channel (signaling channel), each at 64 Kbps. An E1 ISDN PRI transmits 30 B channels, one D channel and one framing channel (synchronization channel), each at 64 Kbps. A standard digital telecommunication service, available in many countries and most of the United States, that allows the transfer of voice and data over T1 or E1 trunks.

Primary Rate Interface: See *PRI*.

primary thread: See *thread*.

process (Linux): the execution of a program. In Linux, process incorporates the concept of an execution environment that includes the contents of memory, register values, name of the current directory, status of files and various other information. Each process is a distinct entity, able to execute and terminate independent of all other processes. A process can be forked/split into a parent process and a child process with separate but initially identical, parent's permissions, working directory, root directory, open files, text, data, stack segments, etc. Each child process executes independently of its parent process, although the parent process may explicitly wait for the termination of one or more child processes.

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.

Process or System Scheduler for Linux: controls the execution of each process or program. This Scheduler enables processes to spawn (create) child processes that are necessary for the operation of the parent process. By default, the Scheduler uses a time-sharing policy that adjusts process priorities dynamically to provide good response time for interactive processes and good throughput for CPU intensive processes. The Scheduler also enables an application to specify the exact order in which processes run. The Scheduler maintains process priorities based on configuration parameters, process behavior and user requests.

PSI: Protocol State Information file used by the PDKRT to define a specific protocol.

PSTN: See *Public Switched Telephone Network*.

Public Switched Telephone Network (PSTN): Refers to the worldwide telephone network accessible to all those with either a telephone or access privileges.

QSIG: QSIG) is a protocol for Integrated Services Digital Network (ISDN) communications based on the Q.931 standard. It is used for signaling between digital private branch exchanges (PBXs). QSIG is employed in voice over IP (VoIP) networks, virtual private networks (VPNs), and high-speed, multi-application networks.

R2 MFC: An international signaling system that is used in Europe, South America and the Far East to permit the transmission of numerical and other information relating to the called and calling subscribers' lines.

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

result value: Describes the reason for an event.

RFU: Reserved for future use.

SCbus: Signal Computing bus. Third generation Time Division Multiplexed (TDM) resource sharing bus that allows information to be transmitted and received among resources over multiple data lines. A hardwired connection between Switch Handlers on SCbus-based products for transmitting information over 1024 time slots to all devices connected to the SCbus.

SCSA: Signal Computing System Architecture. An open-hardware and software standard architecture that incorporates virtually every other standard in PC-based switching. SCSA describes the components and specifies the interfaces for a signal processing system. SCSA describes all elements of the system architecture from the electrical characteristics of the SCbus and SCxbus to the high level device programming interfaces. All signaling is out of band. In addition, SCSA offers time slot bundling and allows for scalability.

SDP: Site Dependent Parameter file used by the PDKRT. Protocol configuration parameters that are user modifible for a specific installation site.

SIT: See *Special Information Tone*.

Special Information Tone (SIT): Detection of an SIT sequence indicates an operator intercept or other problem in completing a call.

SRL (Standard Runtime Library): A Dialogic® library that contains C functions common to all 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.

synchronous function: Synchronous functions block an application or process until the required task is successfully completed or a failed/error message is returned.

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.

synchronous mode: Programming characterized by functions that run uninterrupted to completion. Synchronous functions block an application or process until the required task is successfully completed or a failed/error message is returned.

T1: A digital line transmitting at 1.544 Mbps over 2 pairs of twisted wires. Designed to handle a minimum of 24 voice conversations or channels, each conversation digitized at 64 Kbps. T1 is a digital transmission standard in North America.

T1 robbed bit: A T1 digital line using robbed bit signaling. In T1 robbed bit signaling systems, typically the least significant bit in every sixth frame of each of the 24 time slots is used for carrying dialing and control information. The signaling combinations are typically limited to ringing, hang up, wink and pulse digit dialing.

TBCT: See *Two B Channel Transfer*.

TEI: Terminal Endpoint Identifier (see Recommendations Q.920 and Q.921).

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

termination events: Dialogic® Global Call API 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.

time slot: In a digital telephony environment, a normally continuous and individual communication (for example, someone speaking on a telephone) is (1) digitized, (2) broken up into pieces consisting of a fixed number of bits, (3) combined with pieces of other individual communications in a regularly repeating, timed sequence (multiplexed), and (4) transmitted serially over a single telephone line. The process happens at such a fast rate that, once the pieces

are sorted out and put back together again at the receiving end, the speech is normal and continuous. Each individual pieced-together communication is called a time slot.

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.

Two B Channel Transfer (TBCT): Connects two independent B Channel calls at an ISDN PRI user's interface to each other at the PBX or CO. The ISDN PRI user sends a Facility message to the PBX or CO requesting that the two B Channel calls be connected. If accepted, the user is released from the calls.

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

USID: User Service Identifier.

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.

Vari-A-Bill: Service bureaus can vary the billing rate of a 900 call at any time during the call. Callers select services from a voice-automated menu and each service can be individually priced.

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 Dialogic® Voice API library's **dx_open( )** function.

voice resource: See *voice channel*.

Index

A

abandoned calls 47
alarm flow 102
alarm handling 97
alarm source objects 97
ALARM_SOURCE_ID_NETWORK_ID
    usage 101
alarms 97
    blocking 99
    GCEV_UNBLOCKED event 99
    non-blocking 99
    recovery 99
application-handler thread, Windows 75
ASO 97
asynchronous callback model, Linux 28
asynchronous mode
    Windows 28
asynchronous mode programming
    Linux 27
asynchronous models
    Linux 27
    Windows 28
asynchronous polled model
    Linux 27
asynchronous programming model
    Windows 29
asynchronous with SRL callback 75
asynchronous with SRL callback model
    Windows 29
asynchronous with Win32 synchronization
    Windows 28
asynchronous with Win32 synchronization model 30
asynchronous with Windows® callback
    Windows 28
asynchronous with Windows® callback model
    Windows 29

B

blind call transfer 69
blocking alarms 99
    time slot level 100
    trunk level 100
blocking condition 73

C

call reference number
    multiple 84
call state
    transitions summary 53
call states
    asynchronous termination summary 61
call teardown 61
call termination 61
    asynchronous 62
call transfer
    supervised 67
    unsupervised 69
CRN (Call Reference Number) 22
    lifespan 23

D

data structures
    GC_RTCM_EVTDATA 116
    METAEVENT 73
device handles
    extracting 93
Disconnected state
    transition 62
    transition when alarm occurs 99
drop and insert applications
    programming tips 78

E

error events
    GCEV_TASKFAIL 85
error handling 85
event data in metaevent 73
event handlers 74, 75
    event handler thread 75
    Linux 74
    SRL event handler thread 30
    Windows 29
event mask 64
event notification, asynchronous mode programming 28
event processing, Windows 75

events
  CRN in METAEVENT structure 73
  LDID association 22
  non Global Call events 73
  reason code 73
  retrieving 73
exiting an application
  programming tips 77
extended asynchronous programming model, Windows 28, 30

F

Features
  call control 16
  operation, administration and maintenance 17
firmware 112
firmware module 112

G

gc_BlindTransfer(_) 67
gc_Close(_)
  LDID becomes invalid 22
  programming tips 77
gc_CompleteTransfer(_) 67
gc_DropCall(_) 62
  programming tips 77
gc_GetMetaEvent(_) 27, 28, 29, 30, 73, 75
gc_GetMetaEventEx(_) 30, 31, 73
  caution re. Multiple threads 31
  programming tips 78
gc_GetResourceH(_)
  programming tips 77
gc_HoldCall(_) 65
gc_OpenEx(_)
  LDID assignment 22
gc_ReleaseCallEx(_) 23, 62
  programming tips 77
gc_ResultInfo(_) 73, 85, 99
gc_RetrieveCall(_) 65
GC_RTCM_EVTDATA data structure 116
gc_SetConfigData(_) 64
gc_SetupTransfer(_) 67
gc_SwapHold(_) 67
gc_WaitCall(_) 100
  GCEV_UNBLOCKED event 100
gcerr.h header 85
GCEV_ALARM 100
GCEV_ALARM events 101
GCEV_BLOCKED 99
GCEV_BLOCKED event
  Alarm On condition 99
GCEV_DISCONNECTED event
  asynchronous call termination 62
  sent when alarm occurs 99
GCEV_GETCONFIGDATA_FAIL event 85
GCEV_SETCONFIGDATA event 85
GCEV_TASKFAIL
  error indicating event 74
GCEV_TASKFAIL event 85
GCEV_UNBLOCKED 99
GCEV_UNBLOCKED event
  Alarm Off condition 100
  with gc_WaitCall(_) pending 100
GCEV_UNBLOCKED event for alarm recovery 99
Global Call
  API overview 19
  architecture 18
  call control library overview 20
  major components 15
  product overview 15

I

ID number
  library 21
identifying a call using CRN 22
Idle state
  transition to 62
information retrieval via metaevents 73
internal SRL event handler thread 75

L

LDID (Line Device Identifier) 22
  in METAEVENT structure 73
libraries
  ASCII name string 21
  ID numbers 21
Line Device Identifier (LDID) 22
Linux event handlers 74

M

message/eventing
  Windows 29
METAEVENT data structure 28, 31
  caution re. Multiple threads 31
  retrieval of LDID 22

metaevents 73
multiple threads
    caution re. gc_GetMetaEventEx(_) 31

N

network ASO ID
    usage 101
non-blocking alarms 99
non-signal callback model
    Linux 74
non-signal mode, Linux asynchronous callback model 28
Null state
    call termination 62

P

polled model 27
programming tips
    avoiding performance deterioration in Linux 78
    choosing a programming model in Windows 78
    drop and insert applications 78
    general 77
    SRL-related 78
protocol handler 71
protocol operation
    errors 85

R

resource sharing 23

S

setting up a call 53
signal handlers 64
signal mode, Linux asynchronous callback model 28
sr_enbhdlr(_) 29, 75
SR_MODELTYPE 29, 30, 75
SR_MODELTYPE value 75
sr_NotifyEvt(_) 29
sr_setparm(_) 75
SR_STASYNC 75
sr_waitevt(_) 27, 28, 29, 30, 75
sr_waitevtEx(_) 30, 31
SRL
    event handler thread 30
SRL events 27, 28
    handling 74
SRL handler thread 29
    Windows 29
SRL handler thread, Windows 75
SRL-related programming tips 78
state
    accepted 41
    alerting 55
    connected 41
    dialing 55
    null 42, 55
    offered 42
state diagrams
    asynchronous call tear-down 61
states, call establishment 53
supervised call transfer 67

T

terminating a call
    asynchronous mode 62
termination event 27
termination events 28
tips
    drop and insert applications 78
    general programming 77
    SRL-related 78
transfer
    supervised 67
    unsupervised 67, 69

U

unsolicited event
    synchronous mode 64
unsolicited events
    alarm events 99
unsupervised call transfer 69
user-specified message 29

W

Windows® message handling 29