GlobalCall™ API Software Reference

for UNIX and Windows NT

PRINTED ON RECYCLED PAPER

05-0387-003

COPYRIGHT NOTICE

Part Number: 05-0387-003

Publication Date: March, 1998

Dialogic Corporation
1515 Route 10
Parsippany NJ 07054

Technical Support
Phone: 973-993-1443
Fax: 973-993-8387
BBS: 973-993-0864
Email: CustEng@dialogic.com

For Sales Offices and other contact information, visit our website at **http://www.dialogic.com**

Table of Contents

List of Tables

List of Figures

1. How to Use This Guide

The GlobalCall API (Application Programming Interface) provides a uniform call control interface for developing applications for multiple network interface technologies, for a variety of protocols and for various operating systems. This guide provides developers with an overview of GlobalCall; summarizes GlobalCall features and files; provides application development information and other information as it applies to all technologies and protocols for a host computer operating in a UNIX or a Windows NT environment. Information relating to a particular technology or protocol is found in companion volumes, (see *Appendix D* for a list of Related Publications). Products covered, product terminology conventions and the organization of this guide are described in this chapter.

Where differences exist between the implementation of a GlobalCall application in a UNIX or a Windows NT environment, these differences are indicated by qualifying specific items parenthetically or by presenting separate paragraphs/sections devoted to the implementation within a specific operating system environment. Notable differences for Windows NT include the use of executable threads in contrast to the UNIX parent and child processes, an extended asynchronous programming model and functions to support this model and the capability to dynamically link to specific library or libraries as needed.

1.1. Products Covered By This Guide

The GlobalCall software provides a consistent interface for call control using the following Dialogic products. See *Table 1. Hardware Compatibility Chart* for technology and bus compatibility. See the Release Catalog for your operating system or our web site for the Dialogic products that support GlobalCall applications.

1.2. Organization of this Guide

This guide provides:

- an overview of the Dialogic GlobalCall Network Interface Control API for all technologies for UNIX or Windows NT operating systems
- an overview of the functions used to develop network interface control applications and a detailed description of each of these functions
- a synopsis of the GlobalCall library functions
- a listing of the GlobalCall header files
- a definitive glossary for the terms used to describe GlobalCall.

Chapter 2 presents a product overview describing the compatibility, features and structure of GlobalCall.

Chapter 3 presents an overview of programming models, call states, event handling, event definitions and error handling of the GlobalCall API

Chapter 4 provides an overview of the GlobalCall function library and a tabulated summary of the GlobalCall functions.

Chapter 5 describes the data structures used by selected functions.

Chapter 6 contains a detailed description of each GlobalCall function.

Chapter 7 describes the GlobalCall demonstration programs for inbound and outbound protocols.

Appendix A provides a tabulated summary of the GlobalCall functions and events.

Appendix B provides a tabulated summary of the GlobalCall error codes and result values.

Appendix C provides a listing of the GlobalCall header files.

Appendix D lists related publications for further information on GlobalCall API and other Dialogic products.

A **Glossary** and an **Index** follow the appendices.

2. Product Overview

The GlobalCall software provides a uniform application programming interface for multiple network interface technologies. The GlobalCall API:

- is designed to support a variety of protocols for E-1 CAS, T-1 robbed bit, ISDN, analog loop start and other interfaces
- provides a consistent application interface for the various protocols and technologies
- uses the same input and output parameters at the application level to configure and control the different interfaces.

The core GlobalCall functionality provides a uniform interface for developing applications for multiple network interface technologies, for a variety of protocols and for various operating systems. For example, GlobalCall provides a single **gc_MakeCall( )** function for an E-1 CAS interface, a T-1 robbed bit interface, an ISDN interface, and an analog loop start interface that is capable of handling the different requirements of these signaling systems.

Specific functions and parameters are included within the GlobalCall library to address interface-specific applications such as R2 MFC signaling and ISDN. For example, the **gc_CallAck( )** function will request interface specific services in a manner compatible with the particular network interface handling the call.

This chapter addresses:

- hardware compatibility
- GlobalCall features
- GlobalCall architecture
- Call Control libraries

The compatibility of the GlobalCall API with specific Dialogic system software releases is defined in the Release Notes.

2.1. Hardware Compatibility

The Dialogic network interface boards, bus configurations, and signaling systems currently supported are listed in *Table 1. Hardware Compatibility Chart*. Contact your nearest Dialogic Sales Office or visit our web site for the most up-to-date list of supported products.

Table 1. Hardware Compatibility Chart

Product	Bus	Analog	E-1 CAS	T-1 R.B.	E-1 ISDN	T-1 ISDN
D/41ESC	ISA	Yes				
D/160SC-LS	ISA	Yes				
D/240SC	ISA			Yes		
D/300PCI-E1	PCI		Yes♦			
D/300SC-E1	ISA		Yes		Yes	
D/600SC-2E1	ISA		Yes		Yes	
D/320SC	ISA		Yes			
DTI/300SC	ISA		Yes*		Yes	
DTI/301SC	ISA		Yes*		Yes♦	
D/240PCI-T1	PCI			Yes♦		
D/240SC-T1	ISA			Yes		Yes
D/480SC-2T1	ISA			Yes		Yes
DTI/240SC	ISA			Yes*		Yes
DTI/241SC	ISA			Yes		Yes♦

Product	Bus	Analog	E-1 CAS	T-1 R.B.	E-1 ISDN	T-1 ISDN
VFX/40ESC, VFX/40ESC plus, VFX/40SC	ISA	Yes				

♣ = A voice resource is required for this interface.
♦ = Windows NT only
R.B. = Robbed Bit

2.2. GlobalCall Features

GlobalCall presents a consistent interface across multiple types of signaling systems. The GlobalCall API is call oriented; that is, each call initiated by the application or the network is assigned a Call Reference Number (CRN) for control and tracking purposes. Call handling is independent of the line device over which the call is routed.

Among its many advantages, the call-oriented approach:

- provides a common API for multiple signaling systems
- supports a standard interface for network transactions, including call establishment, call maintenance, call clearing, call signaling and control, and application development
- narrows the differences among the APIs used in different operating system environments

The flexible and convenient development environment provided by GlobalCall is achieved by using the Line Device Identifier (LDID) and the Call Reference Number (CRN), which together enable applications to handle call establishment and teardown consistently across hardware platforms or signaling systems.

2.2.1. Line Device Identifier

The LDID is a unique logical number assigned to a specific resource (e.g., a time slot) or a group of resources within a process by the GlobalCall library.

Minimally, the LDID number will represent a network resource. For example, both a network resource and a voice resource are needed to process a R2 MFC dialing function. Using GlobalCall, a single LDID number is used by the application or thread (Windows NT only) to represent this combination of resources for call control.

A LDID number is assigned to represent physical device(s) that will handle a call, such as a network interface resource, when the **gc_Open( ) or 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 (Windows NT only) can retrieve the LDID number associated with the event by using the **linedev** field of the METAEVENT structure (retrieved using the **gc_GetMetaEvent( )** or the **gc_GetMetaEventEx( )** (Windows NT only) function). The terms line device and LDID are used interchangeably.

2.2.2. Call Reference Number

A CRN is a means of identifying a call on a specific **line device**. A CRN is created by the GlobalCall library when a call is requested by the application or thread (Windows NT only) or by the network.

With the CRN approach, the application or thread (Windows NT only) can access and control the call without any reference to a specific physical port or line device. The CRN is assigned immediately after the **gc_MakeCall( )** function is called or when an incoming call is received. This CRN has a single LDID associated with it (e.g., the line device on which the call was made). However, a single line device may have multiple CRNs associated with it (i.e., more than one call may exist on a given line). At any given instant, each CRN is a unique number within a process. After a call is terminated and the **gc_ReleaseCall( )** function is called to release the resources used for the call, the CRN is no longer valid.

2.2.3. Resource Sharing Across Processes

The CRNs and LDIDs assigned by the GlobalCall API library can **not** be shared among multiple processes. These assigned CRNs and LDIDs remain valid only

within the process invoked. That is, you should not open the same physical device from more than one process (nor from multiple threads in a Windows NT environment) for call control purposes. If either of these conditions occur, unpredictable results may occur.

When using multiple threads in a Windows NT environment, a GlobalCall line device can be opened by calling the **gc_Open( ) or gc_OpenEx( )** function from one thread and then closed by calling the **gc_Close( )** function from a different thread. However, when starting or stopping a configured call control library, the **gc_Start( )** and **gc_Stop( )** functions must be called from the same thread, preferably the primary thread.

2.3. GlobalCall Architecture

GlobalCall provides a common interface to multiple network interface specific libraries (i.e., call control libraries). The GlobalCall software consists of a GlobalCall library that uses a set of call control libraries that support a variety of signaling interfaces and protocols. The GlobalCall library provides the following support for all technologies (analog loop start, E-1 CAS, T-1 robbed bit and ISDN, for example):

- provides a common API for handling different network interfaces
- implements basic functions that are common to all interface-specific libraries in a consistent manner
- translates and routes the application or thread (Windows NT only) requests to the appropriate interface-specific library
- screens the call control libraries from the application or thread (Windows NT only)

Thus, GlobalCall allows application programmers to develop their applications without a detailed knowledge of the underlying technology (how compelled signaling works, how to handle ISDN messages, what signaling bits are used to set up a call, etc.). GlobalCall also facilitates easy porting of an application to meet the requirements of other countries, communication systems and different operating systems. See the *GlobalCall Technology User's Guides* for details about technology specific features.

2.4. Call Control Libraries

Each network technology requires a call control library to provide the interface between the network and the GlobalCall library.

The call control libraries currently supported by the GlobalCall API are:

- ANAPI — the call control library controlling access to analog network interfaces using loop start signaling
- ICAPI — the call control library controlling access to network interfaces using T-1 robbed bit signaling or E-1 CAS
- ISDN — the call control library controlling network interfaces connected to an ISDN network

The application only needs to use the GlobalCall library and does not need to use these lower-level libraries directly. The following diagram, *Figure 1. GlobalCall Architecture*, illustrates how the GlobalCall library uses the call control libraries to access the network.

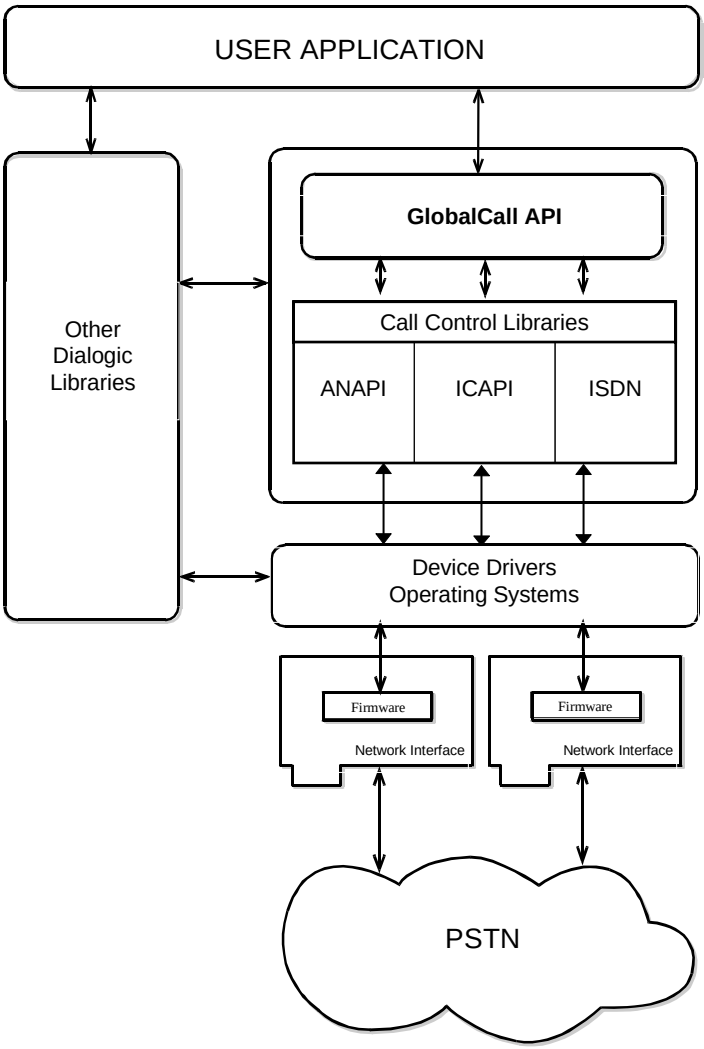

Figure 1. GlobalCall Architecture

2.4.1. Library Terminology for UNIX Environments

Call control libraries must be specifically configured to be used with the GlobalCall library. Such a library is termed a *configured library*. For example, the ANAPI, ICAPI and the ISDN libraries are configured libraries.

For a UNIX application to use the network interface devices supported by the configured libraries, all configured libraries must be linked to the application. For applications in which a particular call control library is not required, a library with a minimal set of internal functions is provided. This library is called a *stub library*. Thus, when an application will only handle a specific technology, a *stub library* may be linked as a substitute for the unused call control library. Using a stub library saves memory, ensures proper startup of the GlobalCall API and enables the application to avoid unresolved external errors while linking. For example, if your application only handles ISDN calls, the ANAPI and ICAPI stub libraries may be linked to your application instead of the ANAPI and ICAPI call control libraries.

All configured call-control libraries, other than stub libraries, start when the **gc_Start( )** function is issued. Once successfully started, these libraries are termed *available libraries*. If a configured non-stub library does not start, the library is termed a *failed library*. A stub library is configured, never starts and is never available.

2.4.2. Library Terminology for Windows NT Environments

Call control libraries must be specifically configured to be used with the GlobalCall library. Such a library is termed a *configured library*. For example, the ICAPI and the ISDN libraries are configured libraries. For a Windows NT application or thread to use the network interface devices supported by a configured library or libraries, the application or thread must call the **gc_Start( )** function which will dynamically link to all configured libraries.

All configured call-control libraries start when the **gc_Start( )** function is called. Once successfully started, these libraries are termed *available libraries*. If a configured library does not start, the library is termed a *failed library*.

2.4.3. Library Information Functions

Each configured call control library is assigned an ID number by GlobalCall. Each library also has a name in 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, the libraries started, 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_CCLibStatus( )**
- **gc_CCLibStatusAll( )**
- **gc_GetVer( )**

3. GlobalCall API

This chapter describes the GlobalCall Application Programming Interface, including:

- UNIX programming models
- Windows NT programming models
- call states
- event handling including event retrieval and alarm handling
- event definitions
- error handling

The GlobalCall API is designed to support the E-1 and T-1 ISDN interfaces, T-1 robbed-bit signaling, E-1 R1 and R2 compelled signaling, analog loop start signaling and other signaling systems. Refer to the *GlobalCall API Software Package Release Notes* for the interfaces and signaling currently supported and to the appropriate *GlobalCall Technology User's Guide* for application development information for a specific interface; see also publications listed in *Appendix D*

The GlobalCall API is call oriented; that is, each call initiated by the application or thread (Windows NT only) or the network is assigned a Call Reference Number (CRN) for control and tracking purposes. Call handling is also 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 or thread (Windows NT only) to address any resource or group of resources using a single device identifier. The flexible and convenient development environment provided by the GlobalCall CRN and LDID enable applications or threads (Windows NT only) to handle call establishment and teardown consistently across hardware platforms or signaling systems.

3.1. UNIX Programming Models

The GlobalCall API provides UNIX synchronous and asynchronous programming models for use in developing your applications . Your call processing will differ

depending on the model used and is discussed later in this chapter in *paragraph 3.3. GlobalCall Call States*.

For UNIX environments, function calls can be handled using:

- a synchronous model or
- an asynchronous model

Applications can use a combination of the UNIX synchronous and asynchronous models.

By usage, the asynchronous and synchronous models are often referred to as the asynchronous and synchronous *operating modes*. This convention is followed in this manual. For detailed information on these models, see the *Standard Runtime Library Programmer's Guide* located in the *Voice Software Reference for UNIX*.

3.1.1. UNIX Synchronous Mode Programming

Synchronous mode programming is 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. Thus, a synchronous function blocks the application and waits for a completion indication from the firmware or driver before returning control to the application. Since further execution is blocked by a synchronous mode function, a separate process is needed for each channel or task managed by the application. A termination event is not generated for a synchronous function.

The synchronous mode can handle multiple calls in a multiline application by structuring the application as a single-line application and then spawning a process for each line required.

3.1.2. UNIX Asynchronous Mode Programming

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 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; and
- a termination event is returned when the requested operation completes.

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

- polled
- callback

When using the UNIX asynchronous polled 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. Event information retrieved 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.

The UNIX asynchronous callback model may be run in signal or non-signal mode. With the callback model, event handlers can be enabled or disabled for specific events on specific devices, see *paragraph 3.8. Event Handling* for details.

3.2. Windows NT Programming Models

The GlobalCall API provides Windows NT synchronous and asynchronous programming models and an extended asynchronous programming model for use in developing your applications . Your call processing will differ depending on the model used and is discussed later in this chapter in *paragraph 3.3. GlobalCall Call States*.

For Windows NT environments, applications can use:

- a synchronous model or a synchronous with SRL callback model
- asynchronous or extended asynchronous models

or a combination of the Windows NT synchronous and asynchronous models; or the Windows NT synchronous and extended asynchronous models.

By usage, the asynchronous and synchronous models are often referred to as the asynchronous and synchronous *operating modes*. This convention is followed in this manual. For detailed information on these programming models, see the *Standard Runtime Library Programmer's Guide* located in the *Voice Software Reference for Windows NT*.

3.2.1. Windows NT Synchronous Mode Programming

Synchronous mode programming is characterized by functions that block thread execution until the function completes or a failed/error message is returned. The operating system can put individual device threads to sleep while allowing other device threads to continue their actions unabated. Thus, a synchronous function waits for a completion indication from the firmware or driver before returning control to the thread. Since further execution is blocked by a synchronous function, a separate thread is needed for each channel or task. When a Dialogic function completes, the operating system wakes up the function's thread so that processing continues. A termination event is not generated for a synchronous function.

The Windows NT synchronous programming model is recommended for less complex applications wherein only a limited number of channels and calls will be handled and processor loading remains light. The synchronous model should be used only for simple and straight flow control applications with only one action per device occuring at any time.

A synchronous model application can handle multiple channels by structuring the application as a single-channel application and then creating a separate synchronous thread for each channel (e.g., for a 60 channel application, the application creates 60 synchronous threads, one thread to handle each of the 60 channels). You would not need event-driven state machine processing because each Dialogic function runs uninterrupted to completion. Since this model calls functions synchronously, it would be less complex than a corresponding asynchronous model application. However, since synchronous applications imply creating a thread or a process for each channel used, these applications tend to slow down the response of the system and to require a high level of system resources (i.e., increases processor loading) to handle each channel. This can

limit maximum device density; thus the synchronous model provides limited scalability for growing systems.

When using the synchronous model, unsolicited events are not processed until the thread calls a Dialogic function such as **gc_GetMetaEvent( )**, **dx_getevt( )** or **dt_getevt( )**. Unsolicited events can be retrieved by creating a separate asynchronous with SRL callback thread, see the *Asynchronous with SRL Callback paragraph* below, (called the combined synchronous and asynchronous model) or by enabling event handler(s) within the application before creating the synchronous threads that handle each channel. For example:

- to use the unsolicited events asynchronous with SRL callback thread approach, the synchronous application would first create an asynchronous thread to handle all unsolicited events and then the application could create synchronous threads, one for each channel, to process the calls on each channel. The asynchronous thread will use the **sr_waitevt( )** function to do a blocking call. When an unsolicited event occurs, the asynchronous unsolicited event-processing thread identifies the event to a device, services the event and notifies the synchronous thread controlling the device of the action taken. When the application runs an unsolicited events asynchronous thread, then the event processing thread internal to the SRL should be disabled by setting the SR_MODELTYPE value of the **sr_setparm( )** function's **parmno** parameter to SR_STASYNC.
- to use the unsolicited event handler(s) approach, the synchronous application would first enable the unsolicited event handler(s) for the device(s) and event(s) and/or for any device, any event. Then the application would create synchronous threads, one for each channel, to process the calls on each channel. When an unsolicited event specified by a enabled event handler occurs, the SRL passes the unsolicited event information to the application. When the application uses the unsolicited event handler(s) approach, then the event processing thread internal to the SRL must be enabled (default). The SRL event processing thread can also be enabled by setting the SR_MODELTYPE value of the **sr_setparm( )** function's **parmno** parameter to SR_MTASYNC.

3.2.2. Windows NT Asynchronous Mode Programming

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; and
- a termination event is returned when the requested operation completes, thus allowing the Dialogic operation (state machine processing) to continue.

In the asynchronous mode, 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 NT environments, the asynchronous models provided for application development further includes:

- combined synchronous and asynchronous programming, see *paragraph 3.2.1. Windows NT Synchronous Mode Programming*
- asynchronous with SRL callback programming (this model can be used with event handlers)
- asynchronous with Windows callback
- asynchronous with Win32 synchronization
- extended asynchronous programming

The asynchronous programming models are recommended for more complex applications that require coordinating multiple tasks. Asynchronous model applications typically run faster than a synchronous model and require a lower level 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 efficiency because multiple channels are handled in a single thread or in a few threads. See *paragraph 3.13. Programming Tips for Windows NT* 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 NT message/eventing mechanism. Asynchronous model applications are typically more complex than a corresponding synchronous model application due to a higher level of resource management (i.e., 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. Event information retrieved 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.

Using Event Handlers in a Windows NT Environment

Typically, in a Windows NT environment, event processing 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 to be used, such as when an application is being ported from UNIX, then you must use the asynchronous with SRL callback model.

The following guidelines apply to using event handlers:

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

By default, when the **sr_enbhdlr( )** function is first called, a thread internal to the SRL is created to service the application enabled event handlers. This SRL

handler thread exists as long as one handler is still enabled. The creation of this internal SRL event handler thread is controlled by the SR_MODELTYPE value of the SRL **sr_setparm( )** function. The SRL handler thread should be:

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

  NOTE: An application-handler thread must NOT call any synchronous functions.

See the *Standard Runtime Library Programmer's Guide* located in the *Voice Software Reference for Windows NT* for the hierarchy (priority) order in which event handlers are called.

Asynchronous with SRL Callback

The asynchronous with SRL callback 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 **sr_enbhdlr( )** created thread 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.
Your 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 some UNIX applications.

Asynchronous with Windows Callback

The asynchronous with windows callback model allows an asynchronous application to receive SRL event notification through the standard Windows NT message handling scheme. This model is used to achieve the tightest possible integration with the Windows NT messaging scheme. Using this model, you could run the entire Dialogic portion of the application 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 GlobalCall events and optionally for non-GlobalCall events, the application must then call the **gc_GetMetaEvent( )** function before servicing the event.

In this model, the SRL event handler thread must be initiated by the application by setting the SR_MODELTYPE value to SR_STASYNC. For detailed information on this programming model, see the *Standard Runtime Library Programmer's Guide* located in the *Voice Software Reference for Windows NT*.

Asynchronous with Win32 Synchronization

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

Extended Asynchronous Programming

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 provide 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 is 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 and the events are processed immediately.

The extended asynchronous model calls the **sr_waitevtEx( )** function for a group of devices (e.g., 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 (i.e., use thread local variables or local variables) or ensure that the METAEVENT data structure is not over written 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.

3.3. GlobalCall Call States

Each call received or generated by GlobalCall is processed through a series of states wherein each state represents the completion of certain tasks and/or the current status of the call. 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) or
- unsolicited events.

For UNIX environments, calls can be handled using the asynchronous model or the synchronous model. For Windows NT environments, calls can be handled using the asynchronous model, extended asynchronous model or the synchronous model. By usage, the asynchronous and synchronous models are often referred to as the asynchronous and synchronous *operating modes*. This convention is followed in this manual. For detailed information on these models, see also the *Standard Runtime Library Programmer's Guide* located in the *Voice Software Reference* for your operating system.

3.4. Asynchronous Mode Operation

In general, when GlobalCall API functions are issued in asynchronous mode, events trigger the transitions between call states. For example, the termination event, GCEV_ANSWERED, causes the call state to change to the Connected state. Likewise, the unsolicited event, GCEV_DISCONNECTED, causes the call state to change to the Disconnected state. The following functions:

- **gc_MakeCall( )** and
- **gc_ReleaseCall( )**

cause the call state to change upon their successful return. For more detail about how a call transitions from state-to-state, see the following paragraphs.

3.4.1. Establishing and Terminating Calls - Asynchronous

Figure 2 illustrates the call states associated with establishing or setting up a call in the asynchronous mode. The call establishment process for outbound calls is shown on the right side of the diagram; the inbound call set up process is shown on the left. All calls start from a Null state. See *Table 2. Call State Definitions* for a summary of the call states.

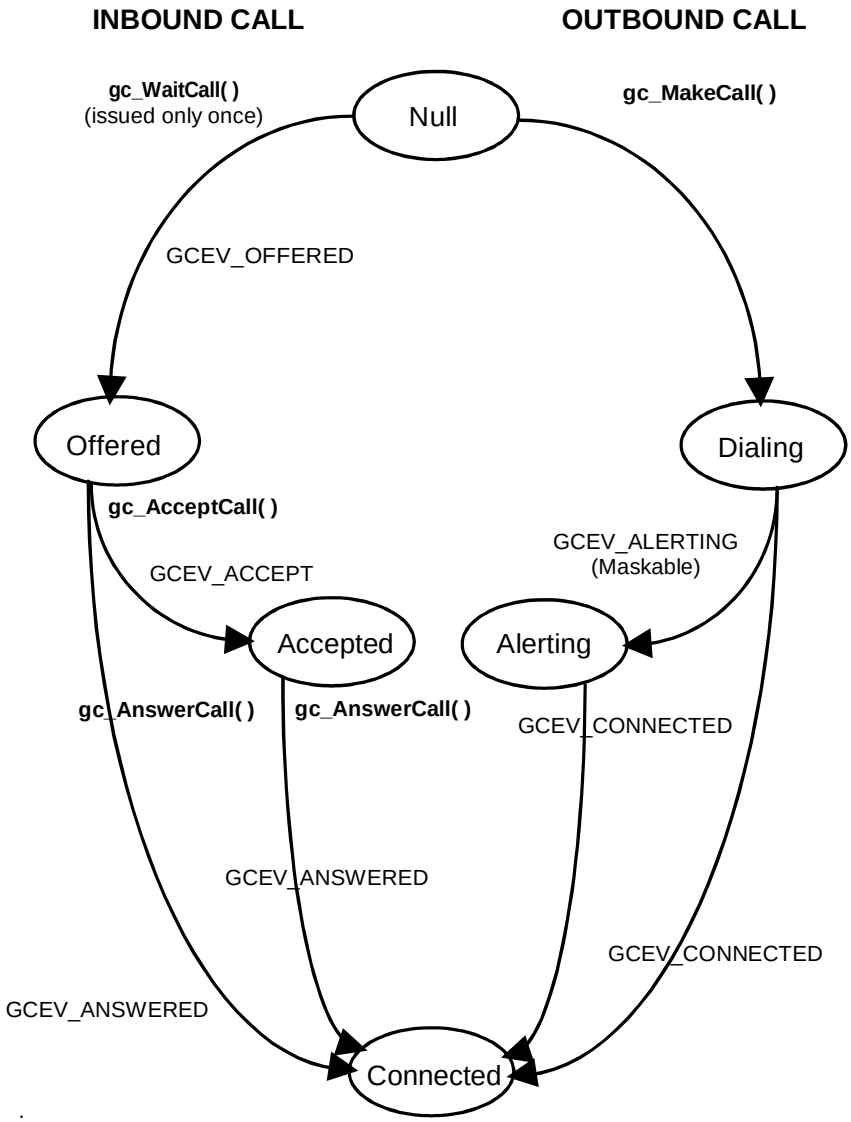

Figure 2. Asynchronous Call Establishment State Diagram

Table 2. Call State Definitions

State	Description
Null	No call is assigned to the device (time slot or line); call released.
Offered	An inbound call from the network is offered to the application or thread (Windows NT only); call received.
Accepted	Indicates an acknowledgment, such as ringing, ringback, etc., to the calling party, that an inbound call is received but not yet connected to the called party; call accepted.
Connected	A connection is established for an inbound or outbound call. Call charge begins.
Dialing	Call establishment is in progress; outbound call request. Dialing information was sent to, and acknowledged, by the network.
Disconnected	Network disconnects the call. Subsequently, the application or thread (Windows NT only) drops the call and releases the CRN and other resources used for the call.
Alerting	The destination was reached and the application or thread (Windows NT only) is waiting for the destination party to answer the call; call alerted sent or received. This state event may be reported to the application or may be masked.
Idle	The call was dropped; call is not active. Subsequently the application or thread (Windows NT only) releases the CRN and other resources used for the call, see *Figure 3. Asynchronous Call Tear-Down State* Diagram.

3.4.2. Inbound Calls - Asynchronous

The application or thread (Windows NT only) issues 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 need only be called once after the line device is opened using the **gc_Open( ) or gc_OpenEx( )** function (unless the **gc_ResetLineDev( )** function was issued). Afterward, the line device will receive calls until closed.

An inbound call is processed as follows, see *Figure 2*. The inbound call from the network is received on the line device specified in the **gc_WaitCall( )** function, thus causing the generation of 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. From the Offered state, the call state changes to either:

- the Connected state or
- the Accepted state.

When the call is to be directly connected, such as to a voice messaging system or the like, 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.

If the application or thread (Windows NT only) 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.

To complete the connection, a **gc_AnswerCall( )** function is issued as described above.

When the call is in the Offered state (after generation of the unsolicited GCEV_OFFERED event) or the Accepted state (before the **gc_AnswerCall( )** function is issued), the application or thread (Windows NT only) may selectively retrieve call information, such as DDI digits (DNIS) and caller ID (ANI). The application or thread (Windows NT only) may also request more dialing information using the **gc_CallAck( )** function.

From the Offered state, the application or thread (Windows NT only) may reject the call by issuing a **gc_DropCall( )** function followed by a **gc_ReleaseCall( )** function, see *Figure 3. Asynchronous Call Tear-Down State* Diagram.

From the Accepted state, not all E-1 CAS 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 attempted, the function will fail and an error is returned. To recover, the application should issue a **gc_AnswerCall( )** function followed by **gc_DropCall( )** and **gc_ReleaseCall( )** functions. See the *GlobalCall Country Dependent Parameters (CDP) Reference* for protocol specific limitations. However, anytime a GCEV_DISCONNECTED event is received in the Accepted state, the **gc_DropCall( )** function can be issued.

Table 3 is an example of a simple inbound call using the asynchronous programming model. The items denoted by a dagger (†) are optional functions/events or maskable events that may be reported to the application for specific signaling protocols. For call scenarios used for a specific signaling protocol, see the *GlobalCall Technology User's Guide* for that protocol.

Table 3. Inbound Call Set-Up (Asynchronous)

Function/Event	Action/Description
gc_WaitCall( )	Issued once after line device opened with **gc_Open( ) or gc_OpenEx( ).**
GCEV_OFFERED	Indicates arrival of inbound call and initiates transition to Offered state.
†gc_GetANI( )	Request caller ID information.
†gc_GetDNIS( )	Retrieves DDI digits received from the network.
†gc_CallAck( )	Request additional call setup information
†GCEV_ACKCALL	Termination event - indicates completion of **gc_CallAck( )** function
†gc_AcceptCall( )	Issued to acknowledge that call was received but called party has not answered
†GCEV_ACCEPT	Termination event - indicates call received, but not yet answered; causes transition to Accepted state.
gc_AnswerCall( )	Issued to connect call to called party (answer inbound call).

Function/Event	Action/Description
GCEV_ANSWERED	Termination event - inbound call connected; causes transition to Connected state.

† = Optional functions and events or maskable events

3.4.3. Outbound Calls - Asynchronous

To initiate an outbound call (see *Figure 2*) 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. This **gc_MakeCall( )** function causes the call state to change 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 the Dialing state, dialing information is sent to and acknowledged by the network. From the Dialing state, the call state changes to either:

- the Connected state or
- the Alerting state.

When the called party immediately accepts the call, such as a call directed to a FAX or voice messaging system or the like, 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.

If the remote end is not ready to answer the call, a GCEV_ALERTING (maskable) event is generated. This event indicates that the called party has not answered the call and that the network is waiting for the called party to complete the connection. This GCEV_ALERTING event changes the call state to the Alerting state. For example:

- for a CAS system, a GCEV_ALERTING event indicates that the remote end is generating ringback and has not answered the call.
- for an ISDN system, a GCEV_ALERTING event indicates that the remote end has sent back an Alerting message.

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; otherwise, 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. For example, if the GCEV_TASKFAIL event is sent, then a problem occurred when placing the call from the local end; whereas, a GCEV_DISCONNECTED event may indicate that the remote end did not answer the call.

When an inbound call arrives while the application is setting up an outbound call, a "glare" condition occurs. 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, the CRN and any resources assigned to the outbound call are released. Subsequently, the GCEV_DISCONNECTED event is generated with a result value indicating that an inbound call took precedence. 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_ReleaseCall( )** function. After release of the outbound CRN, the pending inbound call event is sent to the application.

Table 4 illustrates a simple scenario for making an outbound call using the asynchronous programming model. The items denoted by a dagger (†) are optional functions/events or maskable events that may be reported to the application for specific signaling protocols. For call scenarios used for a specific signaling protocol, see the *GlobalCall Technology User's Guide* for that protocol.

Table 4. Outbound Call Set-up (Asynchronous) Example

Function/Event	Action/Description
†**gc_SetEvtMsk( )**	Specifies the events enabled or disabled for a specified line device.
gc_MakeCall( )	Requests a connection using a specified line device; a CRN is assigned and returned immediately. GCEV_CONNECTED event sent if call connected; otherwise a GCEV_TASKFAIL event is sent.
†GCEV_ALERTING	Remote end was reached but a connection has not been established. When the call is answered, a GCEV_CONNECTED event is sent.
GCEV_CONNECTED	Indicates successful completion of **gc_MakeCall( ).**

† = Optional functions and events or maskable events

3.4.4. Call Termination - Asynchronous

Figure 3 illustrates the call states associated with call termination or call teardown in the asynchronous mode initiated by either call disconnection or failure. See *Table 2. Call State Definitions* for a summary of the call states. A call can be terminated by the application or by the detection of 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.

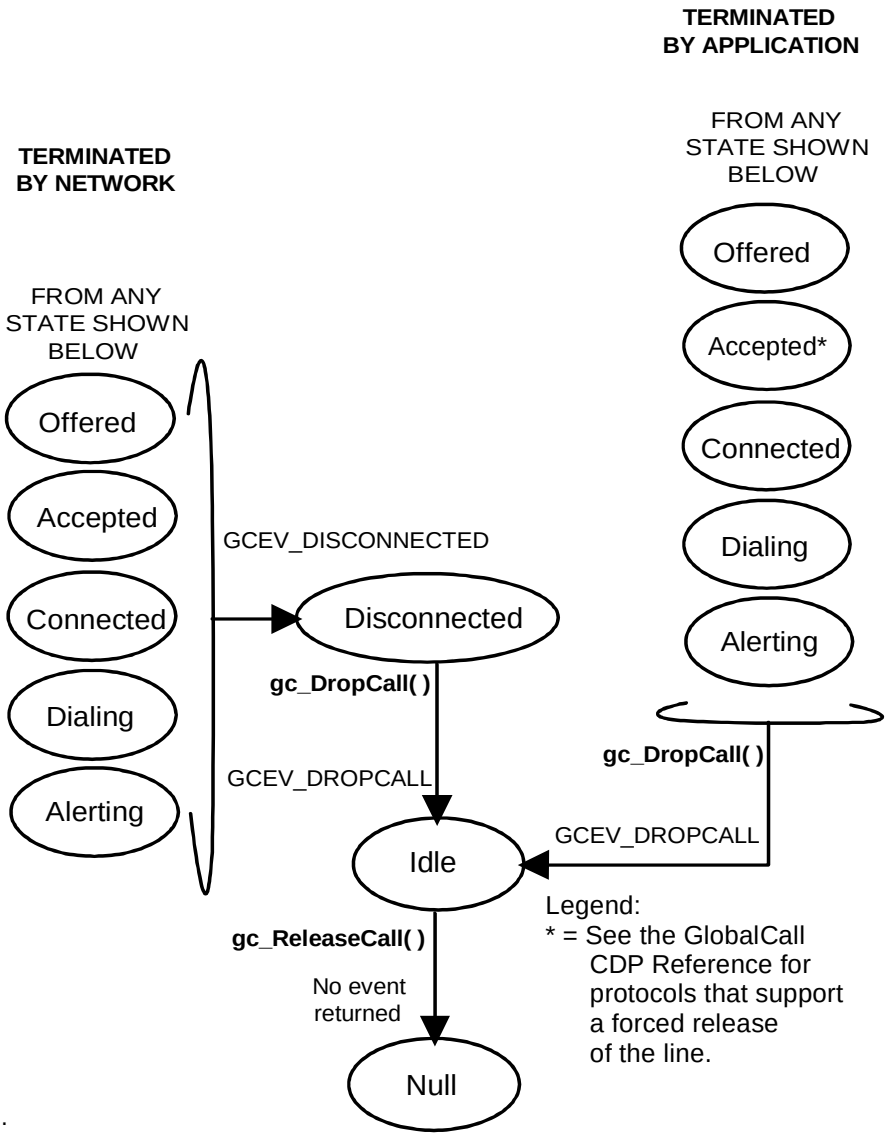

Figure 3. Asynchronous Call Tear-Down State Diagram

The application terminates a call by issuing a **gc_DropCall( )** function that initiates disconnection of the call specified by the CRN. This **gc_DropCall( )** function causes a transition from the current call state to the Idle state. Once in the Idle state, the call has been disconnected and the application must issue a **gc_ReleaseCall( )** function to free the line device for another call. The **gc_ReleaseCall( )** function releases all internal system resources committed to servicing the call and causes a transition to the Null state.

A network call termination is initiated when an unsolicited GCEV_DISCONNECTED event is generated. This event indicates that the call was disconnected at the remote end or that an error was detected that 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 call is disconnected and then waits for a **gc_DropCall( )** function from the application. The **gc_DropCall( )** function is equivalent to "set hook ON." This **gc_DropCall( )** function causes the call state to change to the Idle state. In the Idle state, the **gc_ReleaseCall( )** function releases all internal resources committed to servicing the call and causes a transition to the Null state.

Table 5 presents an asynchronous call termination scenario. For call scenarios used for a specific signaling protocol, check the *GlobalCall Technology User's Guide* for that technology.

Table 5. Call Termination (Asynchronous)

Function/Event	Action/Description
GCEV_DISCONNECTED	Unsolicited event generated when call is terminated by network; initiates transition to Disconnected state.
gc_DropCall( )	Disconnects call specified by CRN. GCEV_DROPCALL event indicates completion of function
GCEV_DROPCALL	Termination event - call disconnected and changes to Idle state.
†gc_GetBilling( )	Retrieve billing information

Function/Event	Action/Description
gc_ReleaseCall( )	Issued to release all resources used for call; network port is ready to receive next call. Causes transition to Null state.

† = Optional functions and events or maskable events

3.5. Synchronous Mode Operation

Functions called in the synchronous mode can be either multitasking or atomic (non-multitasking). A multitasking synchronous function blocks the application until the operation is completed. The function waits for a completion message from the firmware before it terminates.

Atomic synchronous functions typically terminate immediately, return control to the application, and do not cause a call state transition. Most atomic functions receive an (immediate) associated reply message from the firmware at which time the function terminates. Some atomic synchronous functions return information to the application; for example: in response to the **gc_GetDNIS( )** function, the DNIS string is returned and stored in a buffer. Some atomic functions are internal to the driver or firmware and have no need to return any information to the application; for example: the **gc_ReleaseCall( )** function. Note that atomic synchronous functions return information, **not** events.

Figure 4 illustrates the call states associated with establishing or setting up a call in the synchronous mode. The call establishment process for inbound calls and outbound calls is shown. All calls start from a Null state. See *Table 2. Call State Definitions* for a summary of the call states.

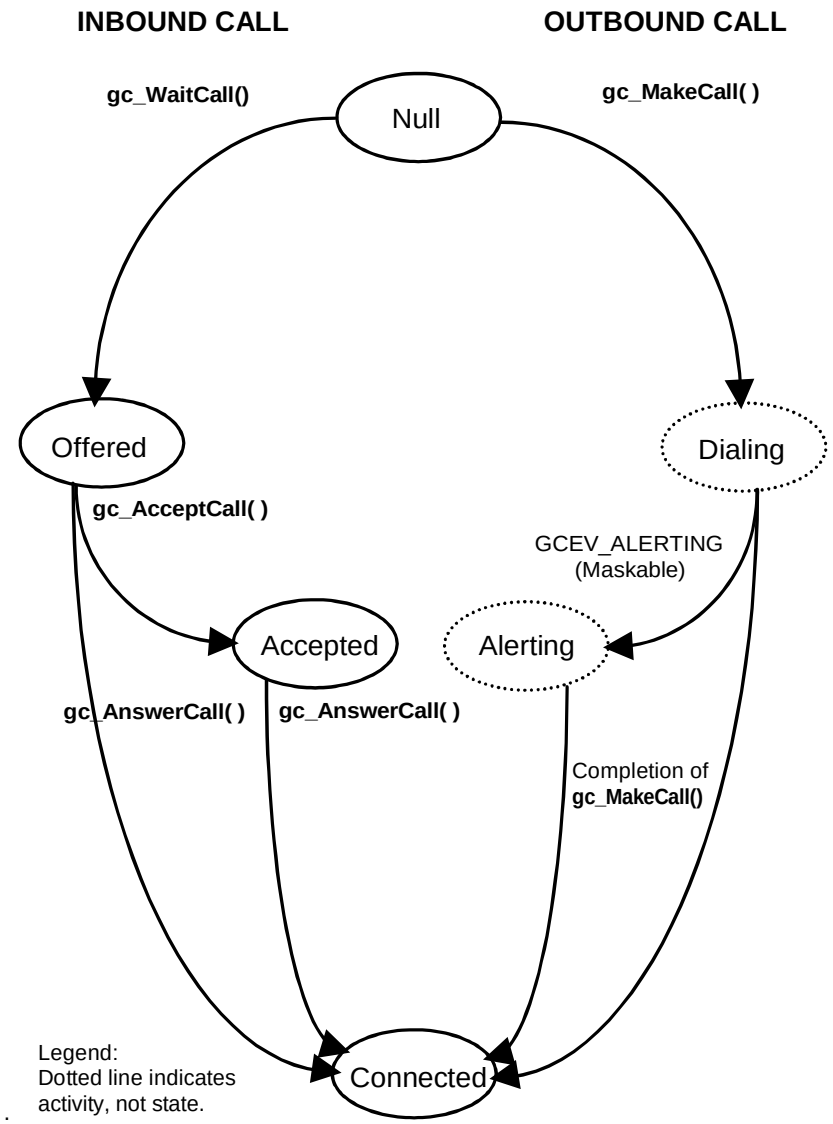

Figure 4. Synchronous Call Establishment Process

3.5.1. Inbound Calls - Synchronous

The application issues a **gc_WaitCall( )** function in the Null state to indicate readiness to accept an inbound call request on the specified line device**.** In the synchronous mode, the **gc_WaitCall( )** function waits for an inbound call for the length of time specified by the **timeout** parameter When the time-out expires, the function fails with an error code, EGC_TIMEOUT, and must be reissued. If the time specified is 0, the function fails unless a call is already pending on the specified line device.

A **gc_WaitCall( )** function waiting for a call to arrive can be stopped (terminated) by issuing a **gc_ResetLineDev( )** function. When the **gc_WaitCall( )** function fails or is stopped, all system resources including the CRN assigned to the call are released. To accept inbound calls, another **gc_WaitCall( )** function must be issued. The application must repeat the poll for incoming calls by issuing a **gc_WaitCall( )** function each time it polls for a call.

An inbound call is processed as follows, see *Figure 4*. The inbound call from the network is received on the specified line device thus causing the call state to change to the Offered state.

In the Offered state, the call may be accepted by the application. From the Offered state, the call state changes to either:

- the Connected state or
- the Accepted state.

When the call is to be directly connected, such as to a voice messaging system or the like, a **gc_AnswerCall( )** function is issued to make the final connection. When the **gc_AnswerCall( )** function is successfully completed, the call changes to the Connected state. At this time, the call is connected to the called party and call charges begin.

If the application 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. When the **gc_AcceptCall( )** function is successfully completed, the call changes to the Accepted state.

To complete the connection, a **gc_AnswerCall( )** function is issued as described above.

When the call is in the Offered state or the Accepted state, the application may selectively retrieve call information, such as DDI digits (DNIS) and caller ID (ANI). The application may also request more dialing information using the **gc_CallAck( )** function.

From the Offered state, the application may reject the call by issuing a **gc_DropCall( )** function followed by a **gc_ReleaseCall( )** function, see *Figure 5. Synchronous Call Tear-Down*.

From the Accepted state, not all E-1 CAS 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 attempted, the function will fail and an error is returned. To recover, the application should issue a **gc_AnswerCall( )** function followed by **gc_DropCall( )** and **gc_ReleaseCall( )** functions. See the *GlobalCall Country Dependent Parameters (CDP) Reference* for protocol specific limitations. However, anytime a GCEV_DISCONNECTED event is received in the Accepted state, the **gc_DropCall( )** function can be issued.

Table 6 is an example of a simple inbound call using the synchronous programming model. The items denoted by a dagger (†) are optional functions/events or maskable events that may be reported to the application for specific signaling protocols. For call scenarios used for a specific signaling protocol, see the *GlobalCall Technology User's Guide* for that protocol.

Table 6. Inbound Call Set-Up (Synchronous)

Function	Action/Description
gc_WaitCall( )	Enables notification of an incoming call after line device opened with **gc_Open( ) or gc_OpenEx( ).**
†gc_GetANI( )	Request ANI information
†gc_GetDNIS( )	Retrieves DDI digits received from the network.
†gc_CallAck( )	Request additional call setup information

Function	Action/Description
†gc_AcceptCall( )	Issued to acknowledge that call was received but called party has not answered
gc_AnswerCall( )	Issued to connect call to called party (answer inbound call).

† = Optional functions

3.5.2. Outbound Calls - Synchronous

To initiate an outbound call (see *Figure 4*) using the synchronous mode, the application issues a **gc_MakeCall( )** function that requests an outgoing call to be made on a specific line device. A CRN is assigned to the call being made on the specific line device. Dialing information is then sent to and acknowledged by the network. When the **gc_MakeCall( )** function is issued in the synchronous mode, the function returns successfully when the call reaches the Connected state. See the *GlobalCall Technology User's Guide* for your technology for valid completion points for the **gc_MakeCall( )** function.

The **gc_SetEvtMsk( )** function specifies the events 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 is not sent to the application. When an event (referred to as a maskable event) is enabled, this event may be received from the network while the **gc_MakeCall( )** function is in progress. Receiving the GCEV_ALERTING event indicates that the called party has not answered the call and that the network is waiting for the called party to complete the connection. For example;

- for a E-1 CAS, T-1 robbed bit or an analog loop start system, a GCEV_ALERTING event indicates that the remote end is generating ringback and has not answered the call.
- for an ISDN system, a GCEV_ALERTING event indicates that the remote end has sent back an Alerting message.

When the call is answered (the remote end makes the connection), the **gc_MakeCall( )** function completes successfully and the call changes to the Connected state.

The application must handle unsolicited events in the synchronous mode, unless these events are masked or disabled. The following unsolicited events, if enabled, require a signal handler:

- GCEV_ALERTING - default is to disable; can be masked.
- GCEV_BLOCKED - default is to enable; can be masked.
- GCEV_UNBLOCKED - default is to enable; can be masked.
- GCEV_DISCONNECTED - default is to enable. Event is not maskable and requires a signal handler.
- GCEV_TASKFAIL - default is to enable. Event is not maskable and requires a signal handler.
- All technology specific unsolicited events (see the *GlobalCall Technology User's Guide* for your technology for details).

If these 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.

If a synchronous **gc_MakeCall( )** function is issued to make an outbound call while an inbound call is in progress, the function fails, and the error value will indicate that an inbound call is in process.

3.5.3. Call Termination - Synchronous

Figure 5 illustrates the call states associated with call termination or call tear-down in the synchronous mode initialized by either call disconnection or failure. See *Table 2. Call State Definitions* for a summary of the call states. A call can be terminated by the application or by the detection of 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.

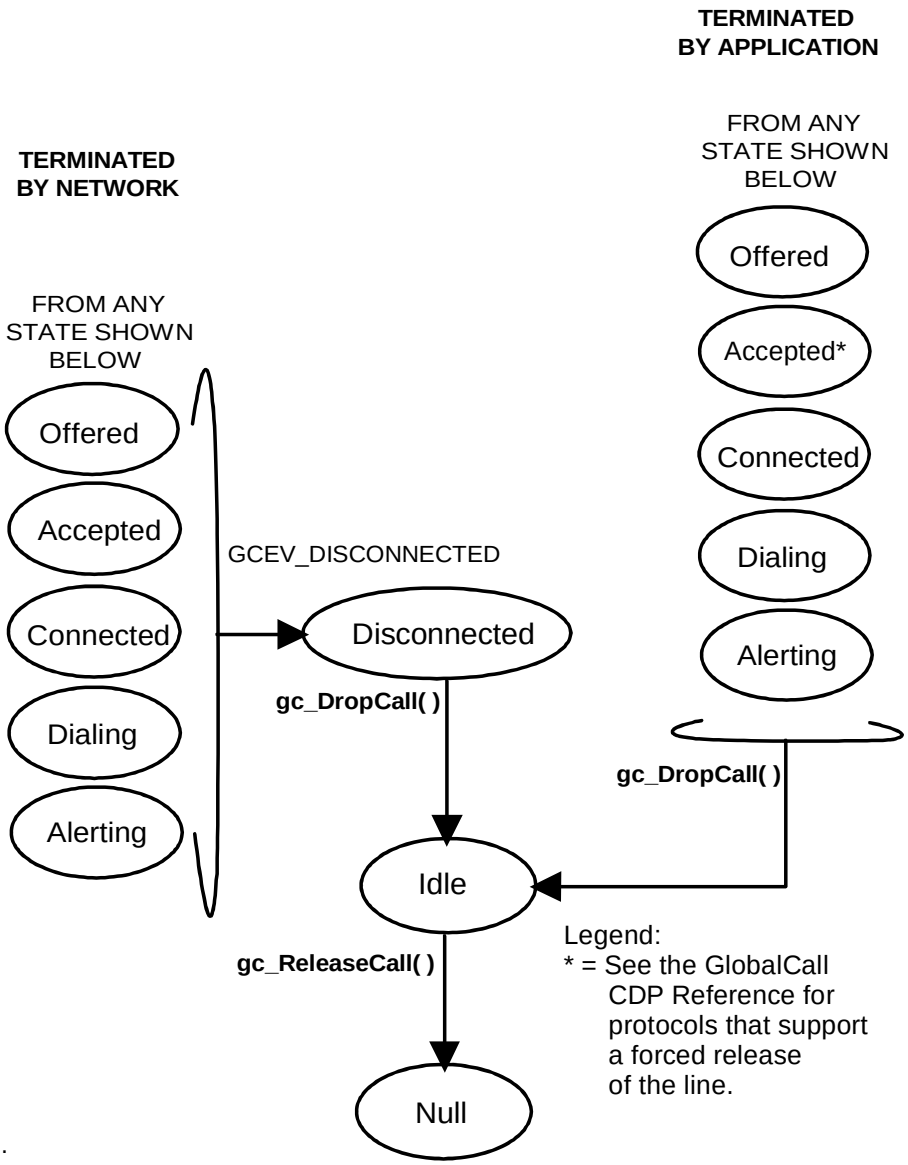

Figure 5. Synchronous Call Tear-Down

The application terminates a call by issuing a **gc_DropCall( )** function that initiates disconnection of the call specified by the CRN. This **gc_DropCall( )** function causes the call to change from the current call state to the Idle state. In the Idle state, the call has been disconnected and the application must issue a **gc_ReleaseCall( )** function to free the line device for another call. This **gc_ReleaseCall( )** function instructs the driver and firmware to release all system resources committed to servicing the call and causes the call state to change to the Null state.

A network call termination is initiated when an unsolicited GCEV_DISCONNECTED event is generated. This event indicates that the call was disconnected at the remote end or that an error was detected that prevented further call processing. The GCEV_DISCONNECTED event causes the call state to change from the current call state to the Disconnected state. In the Disconnected state, the call is disconnected and then waits for a **gc_DropCall( )** function from the application. The **gc_DropCall( )** function is equivalent to "set hook ON." This **gc_DropCall( )** function causes the call state to change to the Idle state. In the Idle state, the **gc_ReleaseCall( )** function instructs the driver and firmware to release all resources committed to servicing the call and causes the call state to change to the Null state.

Table 7 presents a synchronous call termination scenario. For call scenarios used for a specific signaling protocol, check the *GlobalCall Technology User's Guide* for that technology.

Table 7. Call Termination (Synchronous)

Function/Event	Action/Description
GCEV_DISCONNECTED	Unsolicited event generated when call is terminated by network; initiates transition to Disconnected state.
gc_DropCall( )	Disconnects call specified by CRN.
†gc_GetBilling( )	Retrieve billing information
gc_ReleaseCall( )	Issued to release all resources used for call; network port is ready to receive next call. Causes transition to Null state.

† = Optional function

3.6. Routing for UNIX Environments

When using GlobalCall, the standard Dialogic routing functions for routing voice, fax, and other non-network interface resources are used. These routing functions use the device handles of resources such as a voice channel or a network time slot. Two GlobalCall functions, **gc_GetNetworkH( )** and **gc_GetVoiceH( )**, extract the network and voice device handles, respectively, associated with the specified line device. The **gc_GetNetworkH( )** function returns the network device handle for the specified line device. The **gc_GetVoiceH( )** function returns the voice device handle only if the specified line device has a voice or tone resource associated with it (e.g., if a voice channel was specified in the **gc_Open( ) or gc_OpenEx( )** function device name argument and if this channel has remained attached to that line device).

Refer to the appropriate *GlobalCall Technology User's Guide* for technology specific information on routing resources when using the **gc_Open( ) or gc_OpenEx( )** function to specify a voice or tone resource or when using the **gc_Attach( )** function to associate a voice resource with a GlobalCall line device.

3.7. Routing for Windows NT Environments

When using GlobalCall, the standard Dialogic routing functions for routing voice, fax, and other non-network interface resources are used. The **gc_GetNetworkH( )** function returns the network device handle for a specified line device which is then used by the routing functions to route the device. The **gc_GetVoiceH( )** function extracts the voice device handle associated with a specified line device. The **gc_GetVoiceH( )** function returns the voice device handle only if the specified line device has a voice or tone resource associated with it (e.g., if a voice channel was specified in the **gc_Open( )** or **gc_OpenEx( )** function device name argument and if this channel has remained attached to that line device).

Refer to the appropriate *GlobalCall Technology User's Guide* for technology specific information on routing resources when using the **gc_Open( )** or **gc_OpenEx( )** function to specify a voice or tone resource or when using the **gc_Attach( )** function to associate a voice resource with a GlobalCall line device.

3.8. Event Handling

The GlobalCall 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 GlobalCall events that apply to all technologies are described in this section and specific event definitions are described in the next section, *3.9. Event Definitions*. Refer to the appropriate *GlobalCall Technology User's Guide* for technology specific event information.

Each GlobalCall event is classified as an:

- unsolicited event: generated when an alarm is detected or when certain signals are received from the network; or a
- termination event: generated when a function completes (asynchronous mode only).

To enable or disable events on a line device basis, you can use the **gc_SetEvtMsk( )** function.

3.8.1. Event Retrieval

All events are retrieved using the current SRL event retrieval mechanisms (see the *Voice Software Reference for UNIX, Volume 1* or the *Voice Software Reference for Windows NT, Volume 1*, for details), including event handlers. The **gc_GetMetaEvent( )** or **gc_GetMetaEventEx( )** (Windows NT extended asynchronous mode) function maps the current SRL event into a **metaevent**. This 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 GlobalCall events, the structure contains GlobalCall related information (CRN and line device) used by the application. For non-GlobalCall events, the Dialogic device descriptor, the event type, the event data pointer to the variable length data and the length of the variable data are available through the METAEVENT structure. Since event data is present in the metaevent and thus will be stored in the METAEVENT structure, corresponding SRL calls to obtain event information need not be made.

The LDID associated with an event is available from the **linedev** field of the metaevent. The CRN associated with each event is available from the **crn** field of the metaevent (only if the event is CRN related). If the CRN is 0, then the event is not a call related event.

Late events are events that arrive for a released CRN. Late events can occur if the **gc_ReleaseCall( )** function is issued before the application has retrieved all of the termination events. To avoid late events, the application should issue a **gc_DropCall( )** function before issuing the **gc_ReleaseCall( )** 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
- blocking condition
- events sent to the previous user of a CRN could be processed by a later user of the CRN with unexpected results.

The reason or result code for an event is retrieved using the **gc_ResultValue( )** function. The code returned uniquely identifies the cause of the event. Having

retrieved the result value of the event, use the **gc_ResultMsg( )** function to retrieve the ASCII string that describes this result value.

Event Handler for UNIX

An event handler is a user-defined or a GlobalCall API defined function called by the SRL to handle a specific event that occurs on a specified device. The following guidelines apply to UNIX event handlers (For detailed information, see the *Standard Runtime Library Programmer's Guide* located in the *Voice Software Reference for UNIX*):

- 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.

After initiation of an asynchronous function when using the asynchronous signal callback model, the process can receive termination (solicited) or unsolicited events. When an event occurs, the process is interrupted and control is assigned to a central signal handler within the SRL. From this central signal handler, the SRL calls all event handlers that are enabled for that event on that device. After all event handlers are called, control returns to the process at the place where the interrupt occurred and the process continues until notified of the next event.

When using the asynchronous non-signal callback 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 the non-signal callback 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.

Event Handler for Windows NT

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

guidelines listed in *paragraph 3.2.2. Windows NT Asynchronous Mode Programming* apply to Windows NT event handlers (For detailed information, see the *Standard Runtime Library Programmer's Guide* located in the *Voice Software Reference for Windows NT*).

3.8.2. Alarm Handling

GlobalCall alarm events are generated on a line device basis even though alarms occur on a trunk basis. A line device can be associated with an E-1 or T-1 trunk or an individual time slot specified when the **gc_Open( )** or **gc_OpenEx( )** function is issued. Alarm events are unsolicited events sent in addition to other GlobalCall events and do not require any application initiated action. All GlobalCall devices associated with a given E-1 or T-1 trunk on which an alarm occurs will receive a GCEV_BLOCKED event. The blocked event is generated only for the first alarm condition detected. Subsequent alarms on the same trunk will not generate additional blocked events. Until all alarm conditions are cleared, the line device(s) affected by the alarm (i.e., received the GCEV_BLOCKED event) cannot generate or accept calls. Complete alarm recovery is indicated by a GCEV_UNBLOCKED event.

When an alarm occurs while a call is in progress or connected, any calls on the .trunk in the alarm 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 by issuing a **gc_ResultValue( )** function will indicate that an alarm condition occurred. The GCEV_BLOCKED event (Alarm On condition) is also sent to the application to indicate that an alarm occurred. The alarm conditions listed in *Table 8. Alarm Conditions* will generate a GCEV_BLOCKED event. The **gc_ResultValue( )** function may be used to identify the condition that caused the GCEV_BLOCKED event to be generated.

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

When an alarm 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.

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

Table 8. Alarm Conditions

Analog Loop Start Alarms:

- None

E-1 Alarms:

- Bipolar violation count saturation
- CRC4 error count saturation
- Driver performance monitor failure
- Error count saturation
- Initial loss of signal detection
- Received distant multi-frame alarm
- Received frame sync error
- Received loss of sync
- Received multi frame sync error
- Received remote alarm
- Received signaling all 1’s
- Received unframed all 1’s

T-1 Alarms:

- Bipolar eight zero substitution detected
- Bipolar violation count saturation

- Driver performance monitor failure
- Error count saturation
- Frame bit error
- Got a read alarm condition
- Initial loss of signal detection
- Out of frame error, count saturation
- Received blue alarm
- Received carrier loss
- Received loss of sync
- Received yellow alarm

3.9. Event Definitions

The following GlobalCall scenarios briefly describe events common to all protocol interfaces (see the appropriate *GlobalCall Technology User's Guide* for a specific protocol for additional events supported by that protocol):

- inbound call events (*Table 9*),
- outbound call events (*Table 10*),
- disconnect/failure events (*Table 11*) and
- other GlobalCall events (*Table 12* and *Table 13*).

For termination events, the terminated function is listed in the "Terminates" column; termination events only apply when using the asynchronous programming model.

For unsolicited events, 'Unsolicited' appears in the "Terminates" column; unsolicited events apply to both the synchronous and the asynchronous programming models. The referenced parameter, CRN or LDID, is identified for each event in the "Ref" column. If the event is maskable, its default setting is

indicated in the "Terminates" column. Refer to the **gc_SetEvtMsk( )** function description in *Chapter 6. Function Reference* for specific information regarding enabling and disabling events.

Table 9. Inbound Call Events

Event	Terminates	Ref	Description
GCEV_ACCEPT	**gc_AcceptCall( )**	CRN	Call received at remote end, but not yet answered
GCEV_ANSWERED	**gc_AnswerCall( )**	CRN	Call established and enters Connected state
GCEV_ACKCALL	**gc_CallAck( )**	CRN	Indicates termination of **gc_CallAck( )** and that the DDI string may be retrieved by using **gc_GetDNIS( )**
GCEV_OFFERED	Unsolicited	CRN	Inbound call arrived; call enters Offered state.

Table 10. Outbound Call Events

Event	Terminates	Ref	Description
GCEV_ALERTING	Unsolicited (enabled by default)	CRN	Destination party has answered call.
GCEV_CALLSTATUS	Unsolicited	CRN	Indicates that a timeout or a no answer (call control library dependent) condition was returned while the **gc_MakeCall( )** function is active
GCEV_CONNECTED	**gc_MakeCall( )**	CRN	Call is connected

Table 11. Disconnected/Failed Call Events

Event	Terminates	Ref	Description
GCEV_DROPCALL	**gc_DropCall( )**	CRN	Call is disconnected and call enters Idle state
GCEV_DISCONNECTED	Unsolicited	CRN	Call disconnected by remote end.
GCEV_DISCONNECTED	Any request or message rejected by network or that has timed-out	Either CRN or LDID	The error detected prevents further call processing on this call.
GCEV_RESETLINEDEV	**gc_ResetLineDev( )**	LDID	Disconnects any active calls on the line device.

Table 12. ISDN Call Events

Event	Terminates	Ref	Description
GCEV_CALLINFO	Unsolicited	CRN	Generated when an incoming information message is received.
GCEV_CONGESTION	Unsolicited	CRN	Generated when an incoming congestion message is

Event	Terminates	Ref	Description
			received.
GCEV_D_CHAN_STATUS	Unsolicited	LDID	Generated when the status of the D channel changes.
GCEV_DIVERTED	Unsolicited	CRN	Received request to call forward using DPNSS protocol.
GCEV_FACILITY	Unsolicited	LDID	Generated when an incoming facility message is received.
GCEV_FACILITY_ACK	Unsolicited	LDID	Generated when an incoming facility ACK message is received.
GCEV_FACILITY_REJ	Unsolicited	LDID	Generated when an incoming facility reject message is received.
GCEV_HOLDACK	**gc_HoldCall( )**	CRN	Generated when an acknowledgement is sent in response to a hold call message.
GCEV_HOLDCALL	Unsolicited	CRN	Generated when a hold current call message is

Event	Terminates	Ref	Description
			received.
GCEV_HOLDREJ	**gc_HoldCall( )**	CRN	Generated when a hold call request is rejected and the hold call reject message is sent to remote end.
GCEV_ISDNMSG	Unsolicited	CRN	Generated when an incoming unrecognized ISDN message is received.
GCEV_L2BFFRFULL	Unsolicited	CRN	Generated when the incoming layer 2 access message buffer is full. (reserved for future use)
GCEV_L2FRAME	Unsolicited	CRN	Generated when an incoming layer 2 access message is received.
GCEV_L2NOBFFR	Unsolicited	CRN	Generated when no free space is available for an incoming layer 2 access message.
GCEV_NOTIFY	Unsolicited	CRN	Generated when an incoming notify message is received.
GCEV_NSI	Unsolicited	CRN	Generated when

Event	Terminates	Ref	Description
			a Network Specific Information (NSI) message is received using DPNSS protocol.
GCEV_PROCEEDING	Unsolicited (enabled by default)	CRN	Generated when an incoming proceeding message is received.
GCEV_PROGRESSING	Unsolicited (enabled by default)	CRN	Generated when an incoming progress message is received.
GCEV_REQANI	**gc_ReqANI( )**	CRN	Generated when ANI information is received from network.
GCEV_RETRIEVEACK	**gc_RetrieveCall( )**	CRN	Generated when an acknowledgement is sent in response to a retrieve hold call message.
GCEV_RETRIEVECALL	Unsolicited	CRN	Generated when a retrieve hold call message is received.
GCEV_RETRIEVEREJ	**gc_RetrieveCall( )**	CRN	Generated when a rejection message is sent in response to a

Event	Terminates	Ref	Description
			request to retrieve held call.
GCEV_SETBILLING	**gc_SetBilling( )**	CRN	Generated when billing information for the call is acknowledged by the network.
GCEV_SETCHANSTATE	**gc_SetChanState( )** or unsolicited	CRN	Sets operating state of channel. Or if an unsolicited event, generated when the status of the B channel changes or a maintenance message is received from the network.
GCEV_SETUP_ACK	Unsolicited (disabled by default)	CRN	Generated when an incoming setup ACK message is received.
GCEV_TRANSFERACK	Unsolicited	CRN	Generated when an acknowledgement is sent in response to a transfer call to another destination message using DPNSS protocol.

Event	Terminates	Ref	Description
GCEV_TRANSFERCALL	Unsolicited	CRN	Generated when a transfer call to another destination message is received.
GCEV_TRANSFERREJ	Unsolicited	CRN	Generated when a rejection message is sent in response to a request to transfer call to another destination using DPNSS protocol.
GCEV_TRANSIT	Unsolicited	CRN	Generated when a message is sent via a call transferring party to the destination party after a transfer call connection is completed using DPNSS protocol.
GCEV_USRINFO	Unsolicited	CRN	Generated when an incoming User-to-User Information (UUI) message is received.

Table 13. Other GlobalCall Events

Event	Terminates	Ref	Description
GCEV_BLOCKED	Unsolicited (enabled by default)	LDID	Line is blocked and application cannot issue call-related function calls. Retrieve reason for line blockage using **gc_ResultValue( )**.
GCEV_UNBLOCKED	Unsolicited (enabled by default)	LDID	Line is unblocked. Application may issue call-related commands to this line device.
GCEV_SETCHANSTATE	**gc_SetChanState( )**	LDID	Line device is placed in requested state.
GCEV_TASKFAIL	Unsolicited	Either CRN or LDID	An unsolicited error event occurred during the execution of a function.

3.10. Return Value Handling

When a function call returns, the GlobalCall library assigns a return value to indicate to the calling application the success, failure or condition of the results of the call:

- 0 (zero) - returned indicates successful initiation of the function.
- < 0 (zero) - returned indicates the function failed to complete successfully.

When a function fails, a value less than zero is returned. The error code for this failure is retrieved by issuing a **gc_ErrorValue( )** call. This function must be called immediately after the function failed value is returned. Having retrieved the error code for the failure, an ASCII string that describes the reason for the failure may be retrieved by issuing a **gc_ResultMsg( )** function.

NOTE: When a function fails, the value returned is less than zero. Do not test explicitly for a value of −1; future versions of the GlobalCall API may not use −1 as the returned value.

The *gcerr.h* header file contains a comprehensive list of error codes; see listing in *Appendix C*.

3.11. Error Handling

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

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

When a function returns with a value < 0, the error code defining the reason for the failure may be retrieved by calling the **gc_ErrorValue( )** function immediately after the function returns. The **gc_ResultMsg( )** function converts any GlobalCall error code into an ASCII string containing a description of the error.

Call control libraries supported by the GlobalCall 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 also available using the **gc_ErrorValue( )** function.

If an error occurs during execution of an asynchronous function, the GCEV_TASKFAIL 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 *paragraph 3.4.1. Establishing and Terminating Calls - Asynchronous*.

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_ResultValue( )** function.

A call is terminated as shown in *Figure 3. Asynchronous Call Tear-Down State Diagram* and in *Figure 5. Synchronous Call Tear-Down*. 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 also the appropriate *GlobalCall Technology User's Guide* for information on specific protocol errors.

3.12. Programming Tips for UNIX

1. When using GlobalCall functions, the application must use the GlobalCall handles (i.e., line device ID and CRN) to access GlobalCall functions. Do not substitute a network or voice device handle for the GlobalCall line device ID or CRN. If the application needs to use a network or voice device handle for a specific network or voice library call, (e.g., **nr_scroute( )**, **dx_play( )**, etc.), you must use the **gc_GetNetworkH( )** or the **gc_GetVoiceH( )** function to retrieve the network or voice handle, respectively, associated with the specified GlobalCall line device. The **gc_GetVoiceH( )** function is only needed if the voice or tone resource is associated with a GlobalCall line device. If a voice or tone resource is not part of the GlobalCall line device, the device handle returned from the **dx_open( )** call should be used.

2. Do not access the underlying call control libraries directly (i.e., do not issue calls directly to the ANAPI, ISDN or the ICAPI libraries); ALL accesses **must be** via the GlobalCall library.

3. Do not call any network library function directly from your application that may affect the state of the line or the reporting of events (e.g., **dt_settssig( )**, **dt_setevtmsk( )**, or the like).

4. 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.

5. Before exiting an application, perform the following:

   - drop (using the **gc_DropCall( )** function) and release (using the **gc_ReleaseCall( )** function) ALL active calls;

   NOTE: From the Accepted state, not all E-1 CAS 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 attempted, the function will fail and an error is returned. To recover, the application should issue a **gc_AnswerCall( )** function followed by **gc_DropCall( )** and **gc_ReleaseCall( )** functions. See the *GlobalCall Country Dependent Parameters (CDP) Reference* for protocol specific limitations. However, anytime a GCEV_DISCONNECTED event is received in the Accepted state, the **gc_DropCall( )** function can be issued.

   - close all open line devices (using the **gc_Close( )** function).

6. Before issuing a **gc_DropCall( )** function, you must first terminate any voice related function currently in progress. For example, if a play or a record is in progress, then before you can drop the call, issue a stop channel function on that voice channel and then call the **gc_DropCall( )** function to drop the call.

7. When using the *libdti.a* library file, the application must also link with the *libgncf.a* library file.

8. When programming in synchronous mode, performance may deteriorate as the number of synchronous processes increase due the increased UNIX overhead needed to handle these processes. When programming multichannel applications, asynchronous mode programming is likely to provide better performance.

3.12.1. SRL Related Programming Tips for UNIX

1. When a SRL is in signaling mode (SIGMODE), do not call any synchronous mode (i.e., mode=EV_SYNC) GlobalCall function from within a handler registered to the SRL.
2. When a SRL is in signaling mode (SIGMODE) and a GlobalCall function is issued synchronously (i.e., mode=EV_SYNC), then ensure that the application only enables handlers with the SRL to catch the exceptions (i.e., unsolicited events like GCEV_BLOCKED, GCEV_UNBLOCKED or GCEV_DISCONNECTED) instead of enabling wildcard handlers to catch all events. If you enable wildcard handlers, the application may receive unexpected events which should not be consumed.

3.13. Programming Tips for Windows NT

1. 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 interprocess communication and simplifies the coordination of events from many devices.
2. When using GlobalCall functions, the application or thread must use the GlobalCall handles (i.e., line device ID and CRN) to access GlobalCall functions. Do not substitute a network or voice device handle for the GlobalCall line device ID or CRN. If the application or thread needs to use a network or voice device handle for a specific network or voice library call, (e.g., **nr_scroute( )**, **dx_play( )**, etc.), you must use the **gc_GetNetworkH( )** or the **gc_GetVoiceH( )** function to retrieve the network or voice handle, respectively, associated with the specified GlobalCall line device. The **gc_GetVoiceH( )** function is only needed if the voice or tone resource is associated with a GlobalCall line device. If a voice or tone resource is not

part of the GlobalCall line device, the device handle returned from the **dx_open( )** call should be used.

3. Do not access the underlying call control libraries directly (i.e., do not issue calls directly to the ANAPI, ISDN or the ICAPI libraries); ALL accesses **must be** via the GlobalCall library.

4. Do not call any network library function directly from your application or thread that may affect the state of the line or the reporting of events (e.g., **dt_settssig( )**, **dt_setevtmsk( )**, or the like).

5. 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.

6. Before exiting an application, perform the following:

   - drop (using the **gc_DropCall( )** function) and release (using the **gc_ReleaseCall( )** function) ALL active calls;

   NOTE: From the Accepted state, not all E-1 CAS 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 attempted, the function will fail and an error is returned. To recover, the application should issue a **gc_AnswerCall( )** function followed by **gc_DropCall( )** and **gc_ReleaseCall( )** functions. See the *GlobalCall Country Dependent Parameters (CDP) Reference* for protocol specific limitations. However, anytime a GCEV_DISCONNECTED event is received in the Accepted state, the **gc_DropCall( )** function can be issued.

   - close all open line devices (using the **gc_Close( )** function).

7. Before issuing a **gc_DropCall( )** function, you must first terminate any voice related function currently in progress. For example, if a play or a record is in progress, then before you can drop the call, issue a stop channel function on that voice channel and then call the **gc_DropCall( )** function to drop the call.

8. 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.

3.14. Programming Tips for Drop and Insert Applications

When dealing with E-1 CAS or T-1 robbed bit protocols:

- signaling such as line answered is passed to the application as the GCEV_ANSWERED event.
- signaling such as line busy is available to the application as an error code EGC_BUSY or a result value GCRV_BUSY; line no answer as an error code EGC_NOANSWER or a result value GCRV_NOANSWER.
- 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 error code EGC_BUSY or a result value GCRV_BUSY.
- non-backward signaling protocols generate a GCEV_DISCONNECTED event with an error code EGC_BUSY or a result value GCRV_BUSY when time outs or protocol errors occur during dialing.

For a drop and insert application wherein 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 will hang 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. A 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 wherein 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 time out 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:

- use a tone or voice resource to generate a busy tone [60 ipm (impulses per minute)] or to record a busy tone.
- connect this busy tone to the calling party's line or playback the recorded busy tone file.
- then 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:

- use a tone or voice resource to generate a fast busy tone (120 ipm) or to record a fast busy tone.
- connect this fast busy tone to the calling party's line or playback the recorded fast busy tone file.
- then drop and release the calling party's line when a GCEV_DISCONNECTED event is received.

For voice function information, see the *Voice Software Reference* for your operating system.

3.15. Building Applications for UNIX

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

gcerr.h
gclib.h
gcisdn.h (for applications that use ISDN symbols)

When using the ANAPI library or the ICAPI library, the following source file must be compiled by the user and linked to the application:

- for ANAPI library, link *ancountry.c*
- for ICAPI library, link *country.c*

The library files listed in *Table 14. UNIX Files to be* Linked, must be linked to the application IN THE FOLLOWING ORDER:

- *libgc.a* file
- then the library files (or their stub library file) in the order listed
- then the *libdxxx.a*, *libdti.a* and *libsrl.a* files

For each library, either the library files or their corresponding stub library file must be linked. For information on stub libraries, see *paragraph 2.4. Call Control Libraries*.

Table 14. UNIX Files to be Linked

<table>
<tr><td colspan="3">The following library files MUST ALWAYS be linked:
• libgc.a
• libdxxx.a
• libdti.a
• libsrl.a</td></tr>
<tr><td colspan="3">Select the libraries (protocols) to be used with your application and link the files listed below. For libraries not used, link the corresponding stub library file.

NOTE: For each GlobalCall library listed below, either the library files OR the corresponding stub library file MUST be linked to your application.</td></tr>
<tr><td>ICAPI library:</td><td>ANAPI library:</td><td>ISDN library:</td></tr>
<tr><td>• libr2lib.a
• libr2mf.a</td><td>• libatlib.a
• libanalog.a</td><td>• libgcis.a
• libgncf.a</td></tr>
<tr><td colspan="3">Stub library:</td></tr>
<tr><td>• libicapi.a</td><td>• libanapi.a</td><td>• libisdn.a</td></tr>
</table>

3.15.1. Using Only ICAPI Protocols in UNIX Applications

The following object files (located in the */usr/dialogic/ictools* directory) must be linked to the application (i.e., for all installed protocol modules):

- all protocol modules with the format:

  cc_tt_ffff_d.o or cc_tt_d.o

See the *GlobalCall E-1/T-1 Technology User's Guide* for information on the naming convention used for ICAPI protocols.

3.15.2. Using Only Analog Protocols in UNIX Applications

The following object files (located in the */usr/dialogic/ictools* directory) must be linked to the application (i.e., for all installed protocol modules):

- all protocol modules with the format:

  cc_an_ffff_d.o or cc_an_d.o

See the *GlobalCall Analog Technology User's Guide* for information on the naming convention used for analog protocols.

3.16. Building Applications for Windows NT

When building a Windows NT application, the application with its GlobalCall header file includes, is compiled and linked with the *libgc.lib* library file. Thereafter, when you issue a **gc_Start( )** call, the configured library or libraries (e.g., *libgcan.dll* for ANAPI protocols, *libgcr2.dll* for ICAPI protocols, *libgcis.dll* for ISDN protocols, etc.) that you are using are dynamically loaded. If a configured library cannot be found, the GlobalCall API enters an error message in the event logger. When a particular country dependent/specific protocol file(s) (e.g., *br_r2.dll* for Brazil R2 protocol, *us_mf.dll* for U.S. T-1 robbed bit protocol, etc.) is needed, this protocol file(s) is dynamically loaded.

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

gcerr.h
gclib.h

The following library files must be linked to the application:

- *libgc.lib*
- *libdxxmt.lib*
- *libdtimt.lib*
- *libsrlmt.lib*

The *libgcr2.dll* and *libgcis.dll* files are dynamically loaded. The E-1 CAS or T-1 robbed bit protocol modules are also dynamically loaded when needed by the application. These protocol modules use the following naming format:

- cc_tt_ffff_d.dll or cc_tt_d.dll

The analog protocol module(s) is also dynamically loaded when needed by the application. These protocol modules use the following naming format:

- cc_an_ffff_io.dll or cc_an_d.dll

See the *GlobalCall E-1/T-1 Technology User's Guide* or the *GlobalCall Analog Technology User's Guide* for more information on the naming convention used for these protocols.

3.16.1. Compiling and Linking a Windows NT Application

Dialogic Windows NT libraries may be linked and run using Microsoft Visual C+ (2.0 or higher).

3.17. Using Analog, E-1 CAS, T-1 Robbed Bit and ISDN Protocols

To use analog, E-1 CAS and ISDN protocols or analog, T-1 robbed bit and ISDN protocols in the same system, the configuration file settings for each board must reflect the protocol running on that board. In UNIX and Windows NT systems, this configuration file can be updated at installation. Subsequently, the configuration file:

- for UNIX can be updated using a text editor.
- for Windows NT can be updated using the Dialogic Configuration Manager utility.

For example, the configuration file, */usr/dialogic/cfg/dialogic.cfg* for UNIX, for an application using a D/300SC-E1 board (ID = 0) running the Brazil R2 protocol and using a D/300SC-E1 board (ID = 1) running the ISDN CTR4 protocol would be as follows:

```
[Genload - All Boards]
    BLTAddress = D0000
    Dialog/HD  = YES
    BusType    = SCBUS
```

```
[Genload - ID 0]   /* E-1 board running Brazil protocol */
      ParameterFile = br_300.prm
      ClockSource = loop

[Genload - ID 1]   /* E-1 board running ISDN CTR4 protocol */
      ISDNProtocol = ctr4
      ParameterFile = isctr4.prm
      ClockSource = none
```

The E-1 CAS board provides master clock to the SCbus and is loop timed (i.e., taking its clock from the network). The ISDN board receives clock from the SCbus.

For Windows NT, the Dialogic Configuration Manager utility is used to select or update the configuration file for each board.

4. Function Overview

The Dialogic GlobalCall library functions provide the building blocks for creating network interface control applications. An overview of these functions, grouped into the following categories, is presented in this chapter:

- GlobalCall Basic Functions
- Library Information Functions
- Optional Call Handling and Features Functions
- System Controls and Tools Functions
- Interface Specific Functions

Detailed function descriptions are provided in *Chapter 6. Function Reference*.

GlobalCall basic functions may be used to interface with all signaling systems.

The library information functions retrieve the status, names and number of call control libraries.

The GlobalCall optional call handling functions may be used to interface with all signaling systems. These functions provide additional call handling capabilities related to billing and number identification that are not provided by the basic GlobalCall functions. See also the appropriate *GlobalCall Technology User's Guide* for technology specific information.

The GlobalCall system controls and tools functions provide call state, parameter and call control library management capabilities. These functions may be used to interface with all signaling systems.

The GlobalCall interface specific functions are signaling system specific.

All function prototypes are in the *gclib.h* header file.

Table 15. Basic Functions

Function	Description
gc_AnswerCall( )	response to an incoming call (like a "pick up the phone" command)
gc_DropCall( )	disconnects a call; equivalent to a "hang-up"
gc_MakeCall( )	makes an outgoing call
gc_ReleaseCall( )	releases all internal resources for the specified call
gc_WaitCall( )	sets up conditions for processing incoming calls

Table 16. Library Information Functions

Function	Description
gc_CCLibIDToName( )	converts call control library identification code to library name.
gc_CCLibNameToID( )	converts call control library name to library identification code
gc_CCLibStatus( )	retrieves status of the call control library specified
gc_CCLibStatusAll( )	retrieves status information for all call control libraries

Table 17. Optional Call Handling and Features Functions

Function	Description
gc_AcceptCall( )	optional response to an incoming call request; used to indicate "ringing" to the remote end
gc_CallAck( )	enables user to control the response to an incoming call request by retrieving call information from the network. For ISDN PRI applications, **gc_CallAck( )** function is used in overlap receiving operation.
gc_GetANI( )	returns caller identification information
gc_GetBilling( )	gets the charge information for the call, after GCEV_DISCONNECTED event is received or **gc_DropCall( )** function is terminated
gc_GetDNIS( )	gets the DNIS (DDI digits) associated with a specific CRN
gc_GetLinedevState( )	retrieves the status of the specified line device
gc_GetVer( )	returns the version number of the specified software component
gc_SetBilling( )	for protocols that support this feature, sets billing information for the call
gc_SetCallingNum( )	sets the default calling party number on a specific line device; the calling party number thus defined will be used on all subsequent outbound calls
gc_SetChanState( )	sets a channel to the "in-service," "out-of-service," or "in-maintenance" state

Table 18. System Controls and Tools Functions

Function	Description
gc_Close( )	closes a previously opened device and removes the channel from service
gc_CRN2LineDev( )	acquires the line device ID associated with a given CRN
gc_ErrorValue( )	returns the error value/failure reason related to the last GlobalCall function call. To process an error, this function must be called immediately after a GlobalCall function failed.
gc_GetCallState( )	acquires the state of the call associated with the CRN
gc_GetCRN( )	gets the CRN associated with a recently arrived event (such as GCEV_OFFERED)
gc_GetLineDev( )	gets the line device ID associated with a recently arrived event
gc_GetMetaEvent( )	transforms a call control library event (or any SRL event) into a GlobalCall metaevent
gc_GetMetaEventEx( )	(Windows NT extended asynchronous mode only) transforms a call control library event (or any SRL event) into a GlobalCall metaevent. Passes the SRL event handle to the application so that multithreaded applications can be implemented.
gc_GetNetworkH( )	returns network device handle associated with the specified line device
gc_GetParm( )	retrieves the parameter value specified for a line device
gc_GetUsrAttr( )	retrieves the attribute established using

Function	Description
	gc_SetUsrAttr( ) function
gc_Open( )	opens a GlobalCall device and returns a unique line device handle to identify the physical device(s) that carry the call
gc_OpenEx( )	opens a GlobalCall device, sets a user defined attribute and returns a unique line device handle to identify the physical device(s) that carry the call This function can be used in place of the **gc_Open( )** function followed by a **gc_SetUsrAttr( )** function.
gc_ResetLineDev( )	disconnects any active calls on the line device; aborts all calls being setup
gc_ResultMsg( )	retrieves an ASCII string describing the result code
gc_ResultValue( )	returns the cause of an event
gc_SetEvtMsk( )	sets the event mask associated with the specified line device
gc_SetParm( )	sets the default value of parameters used in call setup process
gc_SetUsrAttr( )	sets an attribute defined by the user
gc_Start( )	starts all configured, call control libraries For UNIX applications, non-stub libraries are started.
gc_Stop( )	stops all configured call control libraries started

Table 19. Analog Loop Start Interface Specific Functions

Function	Description
gc_LoadDxParm( )	Sets voice parameters associated with a line device

Table 20. CAS Interface Specific Functions

Function	Description
gc_Attach( )	logically connects a voice resource to a line device
gc_Detach( )	logically detaches a voice resource from the associated line device
gc_GetVoiceH( )	returns the voice device handle associated with the specified call control line device

Table 21. ISDN Interface Specific Functions

Function	Description
gc_CallProgress( )	notifies the network that the connection request is in progress.
gc_GetCallInfo( )	gets information for the call
gc_ReqANI( )	returns the caller's identification, normally included in the ISDN setup message and ANI-on-Demand requests
gc_SetInfoElem( )	enables setting an additional information element in the next outbound ISDN call
gc_SndMsg( )	sends non-call state-related ISDN message to network over the D channel while a call exists
gc_StartTrace( )	start trace and place result in shared RAM
gc_StopTrace( )	stops the trace and closes the file

5. Data Structure Reference

The data structures used by selected GlobalCall functions are described in this chapter. These structures are used to control the operation of functions and to return information. The data structures defined include:

- GC_CALLACK_BLK
- GC_IE_BLK
- GC_MAKECALL_BLK
- METAEVENT
- GC_PARM
- GC_WAITCALL_BLK

The data structure definition is followed by a table providing a detailed description of the fields in the data structure. These fields are listed in the sequence in which they are defined in the data structure.

Refer to the appropriate *GlobalCall Technology User's Guide* for additional technology specific data structures.

5.1. GC_CALLACK_BLK

The GC_CALLACK_BLK structure contains information provided to the **gc_CallAck( )** function regarding the operation to be performed by this function. When using the **gc_CallAck( )** function in E-1 CAS environments, the *dnis service* structure specifies the number of additional DDI digits to be acquired. The structure is defined in the *gclib.h* header file and is also listed below.

```
typedef struct {
   unsigned long type;   /* type of a structure inside following union */
   long rfu;             /* will be used for common functionality */

   union {
      struct {
         int accept;
      } dnis;
      struct {
         int acceptance;
```

```
         LINEDEV linedev;
      } isdn;
      struct {
         long gc_private[4];
      } gc_private;
   } service;            /* what kind of service is requested */
                         /* related to type field */
} GC_CALLACK_BLK,  *GC_CALLACK_BLK_PTR;
```

Table 22. GC_CALLACK_BLK Field Descriptions

Field	Description
type	type of structure inside following union; 'type' specifies the type of service requested by the **gc_CallAck( )** function. For example, to request the retrieval of additional DNIS digits, set 'type' to GCACK_SERVICE_DNIS.
rfu	reserved for future use; must be set to 0.
service	kind of service requested; related to type field
dnis	structure containing the information needed for collecting DDI digits
dnis.accept	indicates type and number of digits to be requested. Set to the number of DDI digits to be collected. Refer to the appropriate *GlobalCall Technology User's Guide* for technology specific information.
isdn	structure containing information for ISDN procedures supported by this function. Refer to the appropriate *GlobalCall Technology User's Guide* for more details.

Field	Description
isdn.acceptance	indicates type of message to be sent to network. Valid values are: • CALL_PROCEEDING to send Proceeding message • CALL_SETUP_ACK to send Setup Acknowledge message
isdn.linedev	the new GlobalCall line device to be used for the call. If set to 0, the channel requested by the network will be used.
gc_private[4]	for internal use by GlobalCall

5.2. GC_IE_BLK

The GC_IE_BLK structure is used to send an Information Element (IE) block to an ISDN interface using the **gc_SetInfoElem( )** or **gc_SndMsg( )** function. Refer to the appropriate *GlobalCall Technology User's Guide* for technology specific information; e.g., for using the **cclib** field.

The structure is defined as follows:

```
typedef   struct   {
   GCLIB_IE_BLK         *gclib;
   void                 *cclib;
}   GC_IE_BLK,   *GC_IE_BLKP;
```

Table 23. GC_IE_BLK Field Descriptions

Field	Description
gclib	pointer to IE information that is common across GlobalCall technologies. Pointer must be set to NULL in this release.
cclib	pointer to IE information that is specific to the call

Field	Description
	control library (technology) being used; refer to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

5.3. GC_MAKECALL_BLK

The pointer to the GC_MAKECALL_BLK structure in the argument list for the **gc_MakeCall( )** function must be set to NULL to use the default value for the call.

The GC_MAKECALL_BLK structure contains information used by the **gc_MakeCall( )** function when setting up a call. The structure is defined as follows:

```
typedef   struct   {
   GCLIB_MAKECALL_BLK   *gclib;
   void                 *cclib;
}   GC_MAKECALL_BLK,   *GC_MAKECALL_BLKP;
```

Table 24. GC_MAKECALL_BLK Field Descriptions

Field	Description
gclib	pointer to information used by the **gc_MakeCall( )** function that is common across technologies. Pointer must be set to NULL in this release.
cclib	pointer to information used by the **gc_MakeCall( )** function that is specific to the call control library (technology) being used; refer to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

5.4. METAEVENT

This structure contains the event descriptor for a metaevent and is defined as follows:

```
typedef   struct   {
   long      magicno;              /* for internal validity check */
                                   /* NOTE: Application calls gc_GetMetaEvent()
                                    * or gc_GetMetaEventEx() (Windows NT) to
                                    * fill in these fields */
                                   /* only valid if an event was returned */
   unsigned long flags;            /* flags field */
   void      *evtdatap;            /* pointer to the event data block -
                                      sr_getevtdatap */
                                   /* will be f(event, cclib) */
   long      evtlen;               /* event length - UNIX or Windows NT sr_getevtlen */
                                   /* May change as libraries are added */
   long      evtdev;               /* event device - sr_getevtdev */
   long      evttype;              /* event type - sr_getevttype */
   LINEDEV   linedev;              /* line device */
   CRN       crn;                  /* crn - if 0, no crn for this event */
   long      rfu2;                 /* reserved for future use */
   void      *usrattr;             /* user attribute associated with linedev */
   int       cclibid;              /* ID of cclib of associated event */
   int       rfu1;                 /* reserved for future use */
} METAEVENT, *METAEVENTP;
```

Table 25. METAEVENT Field Descriptions describes each element used in the metaevent data structure and lists the function that the GlobalCall API used to retrieve the information stored in the associated field. This data structure eliminates the need for the application to issue the listed functions.

Table 25. METAEVENT Field Descriptions

Field	Description	Function Equivalent
magicno	used for internal validity check	None
flags	flags field; GlobalCall flag is set for all GlobalCall events.	None
evtdatap	pointer to the event data block	**sr_getevtdatap( )**
evtlen	event length	**sr_getevtlen( )**
evtdev	event device	**sr_getevtdev( )**
evttype	event type	**sr_getevttype( )**
linedev	line device for GlobalCall events	**gc_GetLineDev( )**
crn	call reference number for GlobalCall events- if 0, no crn	**gc_GetCRN( )**

Field	Description	Function Equivalent
	for this event	
rfu2	reserved for future use	None
usrattr	user assigned attribute associated with the line device.	**gc_GetUsrAttr( )**
cclibid	identification of call control library associated with the event: n = cclib ID number -1 = unknown	
rfu1	reserved for future use	

5.5. GC_PARM

The GC_PARM structure contains information about the call parameter(s) set by the **gc_SetParm( )** function or read by the **gc_GetParm( )** function. The information stored and retrieved is technology dependent; refer to the appropriate *GlobalCall Technology User's Guide* for technology specific information. The structure is defined as follows:

```
typedef   union   {
   short      shortvalue;
   long       longvalue;
   int        intvalue;
   char       charvalue;
   char       *paddress;
   void       *pstruct;
}   GC_PARM;
```

The field of the GC_PARM structure used varies in accordance with the parameter used. The field used for each parameter is listed in *Table 36. Parameter Descriptions, gc_GetParm( ) and gc_SetParm( ).*

5.6. GC_WAITCALL_BLK

The pointer to the GC_WAITCALL_BLK structure in the argument list for the **gc_WaitCall( )** function must be set to NULL in this release.

6. Function Reference

A detailed description of each GlobalCall function included in the gclib.h file, presented in alphabetical order, is contained in this chapter. Unless otherwise indicated, the functions described in this chapter are available for application development in all supported technologies, see the Technology line in the function header table for specific technology applicability. See Appendix C for a listing of the gclib.h file.

6.1. Alphabetical List of Functions

The Dialogic GlobalCall library functions are listed alphabetically in the following paragraphs. The format for each function description is:

Function header	Lists the function name and briefly states the purpose of the function.
Name:	Defines the function name and function syntax using standard C language syntax.
Inputs:	Lists all input parameters using standard C language syntax.
Returns:	Lists all returns of the function.
Includes:	Lists all include files required by the function.
Category:	Lists the category classification of the function.
Mode:	Asynchronous or synchronous
Technology:	Lists the technologies supported by the function: a filled box designates a supported technology. See Release Notes for latest list of supported technologies.
Description paragraph	Provides a description of function operation, including parameter descriptions. A “Termination Event” paragraph describes the event(s) returned to indicate function termination.
Cautions paragraph	Provides warnings and reminders.

Example paragraph	Provides C language coding example(s) showing how the function can be used in application code.
Errors paragraph	Lists specific error codes for each function.
See Also paragraph	Provides a list of related functions.

6.2. Programming Conventions

The GlobalCall functions use the following format:

gc_function(reference, parameter1, parameter2, ..., parameterN, mode)

where:

gc_function:	Function name.
reference:	An input field that directs the function to a specific line device or call when the reference is a CRN or a line device.
parameters:	Input or output fields.
mode:	Input field indicating how the function is executed. Set value to:

- EV_ASYNC for asynchronous mode execution
- EV_SYNC for synchronous mode execution.

NOTE: In the C language coding example listed in the **Example** paragraph, the example code uses the mnemonic GC_SUCCESS as the function return value. GC_SUCCESS is defined in the *gcerr.h* header file to equate to 0.

Name:	int gc_AcceptCall(crn, rings, mode)	
Inputs:	CRN crn	• call reference number
	int rings	• number of rings before return
	unsigned long mode	• async or sync
Returns:	0 if successful	
	<0 if failure	
Includes:	gclib.h	
	gcerr.h	
Category:	optional feature	
Mode:	asynchronous or synchronous	
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit	
	■ Analog	

■ Description

The **gc_AcceptCall( )** function is an optional response to an inbound call request [GCEV_OFFERED event or termination of the **gc_WaitCall( )** function] that acknowledges that the call has been received but is not yet answered (e.g., the phone is ringing). Normally, a **gc_AcceptCall( )** function is not required in most voice termination applications. This function may be used when the application needs more time to process an inbound call request, such as in a drop/insert application in which the outbound dialing process may be time consuming.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
crn:	Call Reference Number
rings:	specifies how long (the number of rings) the protocol handler will wait before notifying the calling entity. (Maximum supported number of rings is 14. Values greater than 14 will be set to 14.) For protocols not using rings, the **rings** parameter is ignored.
mode:	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

Termination Event: In the asynchronous mode, GCEV_ACCEPT event sent to application if successful; GCEV_TASKFAIL event if not successful.

A GCEV_DISCONNECTED event may be reported to the application as an unsolicited event after a **gc_AcceptCall( )** function is issued. When a GCEV_DISCONNECTED event is received, issue **gc_DropCall( )** and **gc_ReleaseCall( )** functions to change the call state to Null.

■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

/*
 * Assume the following has been done:
 *    1. Opened line devices for each time slot on DTIB1.
 *    2. Wait for a call using gc_WaitCall()
 *    3. An event has arrived and has been converted to a metaevent
 *       using gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows NT)
 *    4. The event is determined to be a GCEV_OFFERED event
 */
int accept_call(void)
{
   CRN         crn;             /* Call Reference Number */
   int         gc_error;        /* GlobalCall error code */
   int         cclibid;         /* Call Control Library ID */
   long        cc_error;        /* Call Control Library error code */
   char        *msg;            /* pointer to error message string */


   /*
    * Accept the incoming call.
    */
   crn = metaevent.crn;
   if (gc_AcceptCall(crn, 0, EV_ASYNC) != GC_SUCCESS) {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               metaevent.evtdev, gc_error, msg);
      return(gc_error);
   }

   /*
    * gc_AcceptCall() terminates with GCEV_ACCEPT event.
```

```
    * When GCEV_ACCEPT is received, the state changes to
    * Accepted and gc_AnswerCall() can be issued to complete
    * the connection.
    */
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure or if the GCEV_TASKFAIL event is received, use **gc_ErrorValue( )** or **gc_ResultValue( )**, respectively, and the **gc_ResultMsg( )** function as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_WaitCall( )**
- **gc_AnswerCall( )**

Name:	int gc_AnswerCall(crn, rings, mode)	
Inputs:	CRN crn	• call reference number
	int rings	• number of rings before return
	unsigned long mode	• async or sync
Returns:	0 if successful	
	<0 if failure	
Includes:	gclib.h	
	gcerr.h	
Category	basic call control	
Mode:	asynchronous or synchronous	
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit	
	■ Analog	

■ Description

The **gc_AnswerCall( )** function is equivalent to conventional "set hook off" function in answering an inbound call and must be used to complete the call establishment process. It can be used any time after a GCEV_OFFERED or GCEV_ACCEPT event is received.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
crn:	Call Reference Number
rings:	specifies the number of rings the protocol handler waits before notifying the calling entity. (Maximum supported number of rings is 14. Values greater than 14 will be set to 14.) For protocols not using rings, the **rings** parameter is ignored.
mode:	Set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

Termination Event: In the asynchronous mode, GCEV_ANSWERED event sent to application if successful; GCEV_TASKFAIL event if not successful.

A GCEV_DISCONNECTED event may be an unsolicited event reported to the application after **gc_AnswerCall( )** function is issued.

■ Cautions

The **gc_AnswerCall( )** function can only be called after an inbound call is detected. Otherwise it fails.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

/*
 * Assume the following has been done:
 *    1. Opened line devices for each time slot on DTIB1.
 *    2. Wait for a call using gc_WaitCall()
 *    3. An event has arrived and has been converted to a metaevent
 *       using gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows NT)
 *    4. The event is determined to be a GCEV_OFFERED event
 */
int answer_call(void)
{

   CRN         crn;                /* call reference number */
   int         gc_error;           /* GlobalCall Error */
   int         cclibid;            /* CC Library ID */
   long        cc_error;           /* Call Control Library error code */
   char        *msg;               /* pointer to error message string */

   /*
    * Do the following:
    *    1. Get the CRN from the metaevent
    *    2. Proceed to answer the call as shown below
    */

   crn = metaevent.crn;
   /*
    * Answer the incoming call
    */

   if (gc_AnswerCall(crn, 0, EV_ASYNC) != GC_SUCCESS) {
      /* process error return as shown */
       gc_ErrorValue( &gc_error, &cclibid, &cc_error);
       gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
       printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
                metaevent.evtdev, gc_error, msg);
       return(gc_error);
   }

   /*
    * gc_AnswerCall() terminates with GCEV_ANSWERED event
```

```
    */
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure or if the GCEV_TASKFAIL event is received, use **gc_ErrorValue( )** or **gc_ResultValue( )**, respectively, and the **gc_ResultMsg( )** function as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See also

- **gc_AcceptCall( )**
- **gc_DropCall( )**
- **gc_WaitCall( )**

Name:	int gc_Attach(linedev, voiceh, mode)	
Inputs:	LINEDEV linedev	• GlobalCall line device handle
	int voiceh	• voice device handle
	unsigned long mode	• sync
Returns:	0 if successful	
	<0 if failure	
Includes:	gclib.h	
	gcerr.h	
Category:	interface specific	
Mode:	synchronous	
Technology:	❑ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit	
	■ Analog	

■ Description

The **gc_Attach( )** function attaches a voice resource to the specified line device. By attaching the voice resource, an association is made between the line device and the voice channel. The voice channel specified by the device handle, *voiceh*, will be used to handle related GlobalCall functions requiring a voice resource for that line device.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
linedev:	GlobalCall line device handle
voiceh:	SRL device handle for a voice resource to be attached to the line device. The **voiceh** parameter specifies the voice resource that handles the protocol sections requiring tones (e.g., DTMF dialing or compelled signaling).
mode:	Set to EV_SYNC for synchronous execution
Termination Event:	None

■ Cautions

The **gc_Attach( )** function does **not** perform time slot routing functions. The routing must be done during system configuration or performed by the application using the voice and network routing functions. Alternatively, the **gc_Open( ) or gc_OpenEx( )** function may be used to open, attach and route both the voice and the network resources**.**

If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <gclib.h>
#include <gcerr.h>

int attach(void)
{
   LINEDEV  ldev;                  /* GlobalCall line device handle */
   int      voiceh;                /* Voice channel number */
   int      lineno, brds, tslots;  /* Number of lines, boards and */
                                   /* time slots */
   int      gc_error;              /* GlobalCall Error */
   int      cclibid;               /* CC Library ID */
   long     cc_error;              /* Call Control Library error code */
   char     *msg;                  /* pointer to error message string */

   /*
    * Open line device for 1st network time slot on dtiB1 using inbound
    * Brazilian R2 protocol [E-1 CAS].
    */

   if (gc_Open(&ldev, ":N_dtiB1T1:P_br_r2_i", 0) == GC_SUCCESS) {
      voiceh = dx_open("dxxxB1C1", NULL);
      if (voiceh != -1) {
         if (gc_Attach(ldev, voiceh, EV_SYNC) == GC_SUCCESS) {
            /*
             * Proceed to route the voice and network resources together,
             * and then generate or wait for a call on the line device, 'ldev'.
             */
         } else {
            /* process gc_Attach() error return as shown */
            gc_ErrorValue( &gc_error, &cclibid, &cc_error);
            gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
            printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
                     ldev, gc_error, msg);
            return(gc_error);
         }
      } else {
         /* Process dx_open() error */
      }
```

```
   } else {
      /* process error from gc_Open() using gc_ErrorValue() */
      /* and gc_ResultMsg() */
   }
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_Close( )**
- **gc_Detach( )**
- **gc_GetNetworkH( )**
- **gc_LoadDxParm( )**
- **gc_Open( )** or **gc_OpenEx( )**

Name:	int gc_CallAck(crn, gc_callackp, mode)	
Inputs:	CRN crn	• call reference number
	GC_CALLACK_BLK *gc_callackp	• pointer to additional information for processing call
	unsigned long mode	• async or sync
Returns:	0 if successful <0 if failure	
Includes:	gclib.h gcerr.h gcisdn.h (for applications that use ISDN symbols)	
Category:	optional feature	
Mode:	asynchronous or synchronous	
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ❑ Analog	

■ Description

The **gc_CallAck( )** function provides information about the incoming call to the network or retrieves information from the network about the incoming call. This function is used after receiving a GCEV_OFFERED event (or after the successful completion of the **gc_WaitCall( )** function) and before answering the call. Some services offered by this function are available to all technologies, such as retrieving additional DNIS digits.

When this function is used to request additional DDI digits, use the **gc_GetDNIS( )** function to retrieve the DDI digits.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
crn:	Call Reference Number
gc_callackp:	pointer to the GC_CALLACK_BLK structure where 'type' specifies the type of service requested by the **gc_CallAck( )** function. The GC_CALLACK_BLK data structure and the value for each field are defined and described in *Paragraph 5.1. GC_CALLACK_BLK*.
mode:	Set to EV_ASYNC for asynchronous execution or to

Parameter	Description
	EV_SYNC for synchronous execution. When using ISDN protocols and the **type** field in the GC_CALLACK_BLK data structure is set to GCACK_SERVICE_ISDN, then this **mode** parameter must be set to EV_SYNC.

For example, to use the **gc_CallAck( )** function to collect 4 DDI digits, set:

- **gc_callackp.type** = GCACK_SERVICE_DNIS
- **gc_callackp.service.dnis.accept** = 4

Termination Event: In the asynchronous mode, GCEV_ACKCALL event sent to application if successful; GCEV_TASKFAIL event if not successful.

Depending on the call control library used (e.g., ISDN), the **gc_CallAck( )** function may return either a GCEV_MOREDIGITS or a GCEV_ACKCALL termination event when the **type** field in the GC_CALLACK_BLK data structure is set to GCACK_SERVICE_DNIS.

GCEV_DISCONNECTED event may be an unsolicited event reported to the application after **gc_CallAck( )** function is issued.

■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <memory.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>


/*
 * Assume the following has been done:
```

```
 *    1. Opened line devices for each time slot on DTIB1.
 *    2. Wait for a call using gc_WaitCall()
 *    3. An event has arrived and has been converted to a metaevent
 *       using gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows NT)
 *    4. The event is determined to be a GCEV_OFFERED event
 *
 */
int call_ack(void)
{
   CRN           crn;              /* call reference number */
   GC_CALLACK_BLK callack;         /* type & number of digits to collect */
   char          dnis_buf[GC_ADDRSIZE];  /* Buffer for holding DNIS digits */
   int           gc_error;         /* GlobalCall error code */
   int           cclibid;          /* Call Control Library ID */
   long          cc_error;         /* Call Control Library error code */
   char          *msg;             /* pointer to error message string */

   /*
    * Do the following:
    *    1. Get called party number using gc_GetDNIS() and evaluate it.
    *    2. If three more digits are required by application to properly
    *       process or route the call, request that they be sent.
    */

   memset(&callack, 0, sizeof(callack));

   /*
    * Fill in GC_CALLACK_BLK structure according to protocol
    * or technology used for application, and call gc_CallAck()
    */
   callack.type = GCACK_SERVICE_DNIS;
   callack.service.dnis.accept = GCDG_NDIGIT;
   if (gc_CallAck(crn, &callack, EV_ASYNC) != GC_SUCCESS) {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               metaevent.evtdev, gc_error, msg);
      return(gc_error);
   }

   /*
    * Now collect the remaining digits.
    */
   if (gc_GetDNIS(crn, dnis_buf) != GC_SUCCESS) {
      /* process error from gc_GetDNIS using gc_ErrorValue() and gc_ResultMsg */
   }

   /*
    * Application can answer, accept, or terminate the call at this
    * point, based on the DNIS information.
    */
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure or if the GCEV_TASKFAIL event is received, use **gc_ErrorValue( )** or **gc_ResultValue( )**, respectively, and the **gc_ResultMsg( )** function as described in *Section 3.11. Error Handling* to

retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See also

- **gc_AcceptCall( )**
- **gc_AnswerCall( )**
- **gc_GetDNIS( )**
- **gc_WaitCall( )**

Name:	int gc_CallProgress(crn, indicator)	
Inputs:	CRN crn	• call reference number
	int indicator	• progress indicator
Returns:	0 if successful	
	<0 if failure	
Includes:	gclib.h	
	gcerr.h	
	gcisdn.h	
Category:	interface specific	
Mode:	synchronous	
Technology:	■ ISDN PRI ❑ E-1 CAS ❑ T-1 robbed bit	
	❑ Analog	

■ Description

The **gc_CallProgress( )** function notifies the network that the connection request is in progress. The **gc_CallProgress( )** function is an optional ISDN function that is called after a GCEV_OFFERED event occurs (or after the successful completion of the **gc_WaitCall( )** function) and before a **gc_AcceptCall( )** function is called. Applications may use the **gc_CallProgress( )** function and the message on the D channel to indicate either that the downstream connection is not an ISDN terminal or that inband information is available from the called party.

In the voice terminating mode, this function is not needed. It may be used in a drop and insert configuration where inband Special Information Tone (SIT) or call progress tone is sent in the network direction.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
crn:	Call Reference Number
indicator:	progress indicators listed in *Table 26*.

Table 26. Call Progress Indicators

Code	Description
CALL_NOT_END-TO-END_ISDN	Call is not end-to-end ISDN. In drop and insert configurations, the application may optionally provide this information to the network.
IN_BAND_INFO	In band information or appropriate pattern now available. In drop and insert configurations, the application may optionally notify the network that in-band tones are available.

Termination Event: None.

■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

■ Example

```
/*
 * Assume the following has been done:
 * 1. device has been opened (e.g. :N_dtiB1T1:P_isdn,
 *    :N_dtiB1T2:P_isdn, etc...)
 * 2. gc_WaitCall() has been issued to wait for a call.
 * 3. gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows NT) has been
 *    called to convert the event into metaevent.
 * 4. a GCEV_OFFERED has been detected.
 */


#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>

/*
 * the variable indicator can be assigned one of the two following
 * values CALL_NOT _END_TO_END_ISDN or IN_BAND_INFO.
 */

int call_progress(CRN crn, int indicator)
{
  LINEDEV  ddd;           /* Line device */
```

```
  int      gc_err;         /* GlobalCall Error Code */
  int      cclibid;        /* Call Control library ID */
  long     cclib_err;      /* Call Control Error Code */
  char     *msg;           /* Error Message */

  if(gc_CRN2LineDev(crn, &ddd) != GC_SUCCESS) {
    gc_ErrorValue(&gc_err, &cclibid, &cclib_err);
    gc_ResultMsg(cclibid, cclib_err, &msg);
    printf ("Error: gc_CRN2LineDev ErrorValue: %d - %s\n",
             cclib_err, msg);
    return(cclib_err);
  }

  if(gc_CallProgress(crn, indicator) != GC_SUCCESS) {
    gc_ErrorValue(&gc_err, &cclibid, &cclib_err);
    gc_ResultMsg(cclibid, cclib_err, &msg);
    printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
              ddd, cclib_err, msg);
    return(cclib_err);
  }

  return(0);
}
```

■ Errors

If this function returns a <0 to indicate failure or if the GCEV_TASKFAIL event is received, use **gc_ErrorValue( )** or **gc_ResultValue( )**, respectively, and the **gc_ResultMsg( )** function as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_DropCall( )**
- **gc_WaitCall( )**

Name:	int gc_CCLibIDToName(cclibid, lib_name)	
Inputs:	int cclibid	• ID code of library
	char **lib_name	• pointer to location of library name
Returns:	0 if successful	
	<0 if failure	
Includes:	gclib.h	
	gcerr.h	
Category:	library information	
Mode:	synchronous	
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit	
	■ Analog	

■ Description

The **gc_CCLibIDToName( )** function converts call control library ID to name of call control library. The library name associated with the **cclibid** library identification parameter is stored in a string designated by the **lib_name** parameter.

Parameter	Description
cclibid:	identification number of call control library. If a library name is not associated with this parameter, then NULL is returned.
lib_name:	name of the call control library associated with the **cclibid** parameter. Possible call control library names include *ICAPI* and *ISDN*.

Termination Event: None.

■ Cautions

Do not overwrite the ***lib_name** pointer as it points to private internal GlobalCall data space.

Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

int cclibid_to_name(int cclibid, char **lib_name)
{
   int            gc_error;         /* GlobalCall error code */
   int            sub_cclibid;      /* Call Control Library ID */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* pointer to error message string */

   if (gc_CCLibIDToName(cclibid, lib_name) != GC_SUCCESS) {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &sub_cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error converting library id %d to library name\n", cclibid);
      printf ("Error = %s\n", msg);
      return(gc_error);
   }
   return(0);
}
```

Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

See Also

- **gc_CCLibNameToID( )**

Name: int gc_CCLibNameToID(lib_name, cclibidp)

Inputs: char *lib_name
- name of library

int *cclibidp
- pointer to location of library identification code

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: library information

Mode: synchronous

Technology: ■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ■ Analog

■ Description

The **gc_CCLibNameToID( )** function converts call control library name to ID code. The library identification code associated with the call control library, **lib_name**, is written into ***cclibidp**.

Parameter	Description
lib_name:	name of the call control library whose library ID is to be retrieved. If a library identification code is not associated with this parameter, then a value <0 is returned. Possible library names include *ICAPI* and *ISDN*.
cclibidp:	pointer to identification code of call control library.

Termination Event: None.

■ Cautions

None

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
```

```
#include <gclib.h>
#include <gcerr.h>

int cclibname_to_id(char *lib_name, int *cclibidp)
{
   int            gc_error;         /* GlobalCall error code */
   int            cclibid;          /* Call Control Library ID */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* pointer to error message string */

   if (gc_CCLibNameToID(lib_name, cclibidp) != GC_SUCCESS) {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error converting library name %d to library ID\n", cclibid);
      printf ("Error = %s\n", msg);
      return(gc_error);
   }
   return(0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_CCLibIDToName( )**

Name: int gc_CCLibStatus(cclib_name, cclib_info)
Inputs: char *cclib_name • name of call control library
int *cclib_info • status of call control library
Returns: 0 if successful
<0 if failure
Includes: gclib.h
gcerr.h
Category: library information
Mode: synchronous
Technology: ■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit
■ Analog

■ Description

The **gc_CCLibStatus( )** function retrieves status of call control library specified by the **cclib_name** parameter. Status of a library can be available, configured, failed or stub. This status information is stored in ***cclib_info**.

Parameter	Description
cclib_name:	name of the call control library; valid names include *ICAPI* and *ISDN*. The string must be set to one of these names and terminated by a NULL.
cclib_info:	pointer to location of status information. The status information is a bitmask with either an available, configured or stub mask set (these masks are mutually exclusive) and/or a failed mask:
	• GC_CCLIB_AVL — available library (started successfully)
	• GC_CCLIB_CONFIGURED — configured library
	• GC_CCLIB_FAILED — library failed to start
	• GC_CCLIB_STUB — stub library (cannot be started)

Termination Event: None.

■ Cautions

None

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

int print_cclib_status(char *lib_name)
{
   int            lib_status;       /* state of call control library */
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */

   if (gc_CCLibStatus(lib_name, &lib_status) == GC_SUCCESS) {
      printf("cclib %s status:\n", lib_name);
      printf("   configured:  %s\n",
             (lib_status & GC_CCLIB_CONFIGURED) ? "yes" : "no");
      printf("   available:  %s\n",
             (lib_status & GC_CCLIB_AVL) ? "yes" : "no");
      printf("   failed:  %s\n",
             (lib_status & GC_CCLIB_FAILED) ? "yes" : "no");
      printf("   stub:  %s\n",
             (lib_status & GC_CCLIB_STUB) ? "yes" : "no");
   } else {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error getting gc_CCLibStatus:  ErrorValue: %d - %s\n",
               gc_error, msg);
      return(gc_error);
   }
   return(0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_CCLibStatusAll( )**
- **gc_Start( )**

Name: int gc_CCLibStatusAll(cclib_status)
Inputs: GC_CCLIB_STATUS *cclib_status • pointer to location of library status information
Returns: 0 if successful
<0 if failure
Includes: gclib.h
gcerr.h
Category: library information
Mode: synchronous
Technology: ■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ■ Analog

■ Description

The **gc_CCLibStatusAll( )** function retrieves status of all call control libraries. Information returned includes the number and names of the available, configured, failed and stub call control libraries. The GlobalCall library is not a call control library and is therefore not counted.

Parameter	Description
cclib_status:	pointer to the GC_CCLIB_STATUS structure, see below for details. Possible library names include *ICAPI* and *ISDN*.

The GC_CCLIB_STATUS structure is defined as follows:

```
typedef struct {
        int         num_avllibraries;
        int         num_configuredlibraries;
        int         num_failedlibraries;
        int         num_stublibraries;
        char        **avllibraries;
        char        **configuredlibraries;
        char        **failedlibraries;
        char        **stublibraries;
} GC_CCLIB_STATUS,   *GC_CCLIB_STATUSP;
```

Table 27. GC_CCLIB_STATUS Field Descriptions

Field	Description
num_avllibraries	returns the number of available call control libraries

Field	Description
num_configuredlibraries	returns the number of configured call control libraries
num_failedlibraries	returns the number of failed (did not start) call control libraries
num_stublibraries	returns the number of stub libraries
avllibraries	returns the name(s) of the available libraries in a string terminated with a NULL; for example if both the *ICAPI* and *ISDN* call control libraries are available, then: avllibraries[0] = "ICAPI" avllibraries[1] = "ISDN"
configuredlibraries	returns the name(s) of the configured libraries in a string terminated with a NULL
failedlibraries	returns the name(s) of the failed libraries in a string terminated with a NULL
stublibraries	returns the name(s) of the stub libraries in a string terminated with a NULL

Termination Event: None.

■ Cautions

If any of the **num_*** fields is 0, then the corresponding ***libraries** field is NULL; e.g., if the **num_avllibraries** field is 0, then the **avllibraries** is NULL.

Do not overwrite the fields that are pointers to strings as these point to private internal GlobalCall data space.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
```

```
int print_all_avl_libraries(void)
{
   int             n;
   int             ret;              /* function return code */
   GC_CCLIB_STATUS cclib_status;     /* cclib information */
   int             cclibid;          /* cclib id for gc_ErrorValue() */
   int             gc_error;         /* GlobalCall error code */
   long            cc_error;         /* Call Control Library error code */
   char            *msg;             /* points to the error message string */


   if (gc_CCLibStatusAll(&cclib_status) == GC_SUCCESS) {
      for (n = 0; n < cclib_status.num_avllibraries; n++) {
         printf("Next available library is: %s\n",
                 cclib_status.avllibraries[n]);
      }
   } else {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error getting gc_CCLibStatusAll:  ErrorValue: %d - %s\n",
               gc_error, msg);
      return(gc_error);
   }
   return(0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_CCLibStatus( )**
- **gc_Start( )**

Name:	int gc_Close(linedev)
Inputs:	LINEDEV linedev • GlobalCall line device handle
Returns:	0 if successful <0 if failure
Includes:	gclib.h gcerr.h
Category:	system control and tools
Mode:	synchronous
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ■ Analog

■ Description

The **gc_Close( )** function closes a previously opened device. The application can no longer access the device via the **linedev** parameter and inbound call notification is disabled. Other devices will be unaffected.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
linedev:	GlobalCall line device to close

Termination Event: None.

■ Cautions

The **gc_Close( )** function only affects the link between the calling process and the device. Other processes and devices are unaffected.

If a voice resource is attached to the **linedev** device, the voice resource will be closed by the GlobalCall API. To keep the voice resource open for other operations, use the **gc_Detach( )** function to detach the voice resource from the GlobalCall device before issuing the **gc_Close( )** function.

The **gc_Close( )** function should be issued while the line device is in the Null state.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

#define MAXCHAN   30                /* max. number of channels in system */
/*
 * Data structure which stores all information for each line
 */
struct linebag {
   LINEDEV  ldev;                   /* GlobalCall line device handle */
   CRN      crn;                    /* GlobalCall API call handle */
   int      state;                  /* state of first layer state machine */
} port[MAXCHAN+1];

struct linebag *pline;              /* pointer to access line device */

int close_line_device(int port_num)
{
   LINEDEV     ldev;            /* GlobalCall line device handle
   int         gc_error;        /* GlobalCall error code */
   int         cclibid;         /* Call Control Library ID */
   long        cc_error;        /* Call Control Library error code */
   char        *msg;            /* points to the error message string */

   /* Find info for this time slot, specified by 'port_num' */
   /* (Assumes port_num is valid) */
   pline = port + port_num;
   ldev = pline -> ldev;
   /*
    * close the line device to remove the channel from service
    */
   if (gc_Close(ldev) != GC_SUCCESS) {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error closing linedev 0x%lx, \"%s\"\n", ldev, msg);
      return(gc_error);
   }
   return(0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

See Also

- **gc_Attach( )**
- **gc_Detach( )**
- **gc_Open( )** or **gc_OpenEx( )**

Name:	int gc_CRN2LineDev (crn, linedevp)
Inputs:	CRN crn • call reference number LINEDEV *linedevp • pointer to a location to store linedev
Returns:	0 if successful <0 if failure
Includes:	gclib.h gcerr.h
Category:	system control and tools
Mode:	synchronous
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ■ Analog

■ Description

The **gc_CRN2LineDev( )** function is a utility function that matches a CRN to its line device ID. This function returns the line device identification associated with the specified CRN.

Parameter	Description
crn:	Call Reference Number
linedevp:	pointer to the location where the output LINEDEV identification code will be stored. The line device is created when the function **gc_Open( )** or **gc_OpenEx( )** is called.

Termination Event: None.

■ Cautions

A CRN is valid only during the call until the **gc_ReleaseCall( )** function has been issued.

Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

int crn_to_linedev(CRN crn, LINEDEV *ldevp)
{
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */

   if (gc_CRN2LineDev(crn, ldevp) != GC_SUCCESS) {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on converting CRN to linedev \"%s\"\n", msg);
      return(gc_error);
   }
   return(0);
}
```

Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

See Also

- None

Name:	int gc_Detach(linedev, voiceh, mode)	
Inputs:	LINEDEV linedev	• GlobalCall line device handle
	int voiceh	• voice device handle
	unsigned long mode	• sync
Returns:	0 if successful	
	<0 if failure	
Includes:	gclib.h	
	gcerr.h	
Category:	interface specific	
Mode:	synchronous	
Technology:	❑ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit	
	■ Analog	

■ Description

The **gc_Detach( )** function is used to logically detach a voice resource from the line device. This breaks any association between the line device and the resource, which would have been attached previously to the line device using the **gc_Attach( )** function.

When a **gc_Close( )** function closes a line device, any attached voice resource is closed automatically. To keep the voice device open, first, issue a **gc_Detach( )** function and then issue the **gc_Close( )** function. This will disassociate the voice device from the line device.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
linedev:	GlobalCall line device handle
voiceh:	SRL device handle of the voice resource to be detached from the call control line device
mode:	set to EV_SYNC for synchronous execution

Termination Event: None.

■ Cautions

The **gc_Detach( )** function does **not** perform any routing or unrouting function. Routing must be performed using the voice and network routing functions.

If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

/*
 * Assume the following has been done:
 *    1. The line device (ldev) has been opened, specifying a
 *       network time slot and a protocol. For example, 'devicename
 *       could be ":N_dtiB1T1:P_br_r2_i:V_dxxxB1C1" [E-1 CAS]
 *    2. The voice and network resources have been routed together
 *    3. Voice resource is no longer needed for this line device
 */

/* detaches the ldev's voice handle from ldev */
int detach(LINEDEV ldev)
{
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */
   int            voiceh;           /* Voice handle attached to ldev */

   if (gc_GetVoiceH(ldev, &voiceh) == GC_SUCCESS) {
      if (gc_Detach(ldev, voiceh, EV_SYNC) != GC_SUCCESS) {
         /* process error return as shown */
         gc_ErrorValue( &gc_error, &cclibid, &cc_error);
         gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
         printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
                  ldev, gc_error, msg);
         return(gc_error);
      }

      /*
       * Application should now unroute the voice and network resources from
       * each other (using functions like nr_scunroute() or sb_unroute() to
       * complete the disassociation of them from each other.
       */
   } else {
      /* Process gc_GetVoiceH() error */
   }
   return (0);
}
```

Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

See Also

- **gc_Attach( )**
- **gc_Close( )**
- **gc_Open( ) or gc_OpenEx( )**

Name:	int gc_DropCall(crn, cause, mode)	
Inputs:	CRN crn	• call reference number
	int cause	• reason to drop call
	unsigned long mode	• async or sync
Returns:	0 if successful	
	<0 if failure	
Includes:	gclib.h	
	gcerr.h	
Category:	basic call control	
Mode:	asynchronous or synchronous	
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit	
	■ Analog	

■ Description

The **gc_DropCall( )** function disconnects a call specified by the CRN and enables inbound calls to be detected internally to GlobalCall on the line device. The application will not be notified of the call until after the **gc_ReleaseCall( )** function is issued .

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
crn:	Call Reference Number
cause:	indicates reason for disconnecting or rejecting a call. See *Table 28* for a list of possible causes and refer to the appropriate *GlobalCall Technology User's Guide* for valid and/or additional causes for your specific technology.
mode:	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

Table 28. gc_DropCall( ) Causes

Cause‡	Description
GC_CALL_REJECTED	Call was rejected
GC_CHANNEL_UNACCEPTABLE	Channel is not acceptable
GC_DEST_OUT_OF_ORDER	Destination is out of order
GC_NETWORK_CONGESTION	Call dropped due to traffic volume on network
GC_NORMAL_CLEARING	Call dropped under normal conditions
GC_REQ_CHANNEL_NOT_AVAIL	Requested channel is not available
GC_SEND_SIT	Special Information Tone
GC_UNASSIGNED_NUMBER	Requested number is unknown
GC_USER_BUSY	End user is busy

‡ Refer to the appropriate *GlobalCall Technology User's Guide* for valid and/or additional causes for your specific technology.

Termination Event: In the asynchronous mode, GCEV_DROPCALL event is sent to the application; otherwise, a GCEV_TASKFAIL event is sent.

A GCEV_DISCONNECTED event may be reported to the application as an unsolicited event after the **gc_DropCall( )** function issues.

■ Cautions

The **gc_DropCall( )** function does not release a CRN. Therefore, the **gc_ReleaseCall( )** function must always be used after a **gc_DropCall( )** function. Failure to do so will cause a blocking condition and may cause memory problems due to memory being allocated and not being released.

Before issuing a **gc_DropCall( )** function, you must first terminate any voice related function currently in progress. For example, if a play or a record is in

progress, then before you can drop the call, issue a stop channel function on that voice channel and then call the **gc_DropCall( )** function to drop the call.

From the Accepted state, not all E-1 CAS 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 attempted, the function will fail and an error is returned. To recover, the application should issue a **gc_AnswerCall( )** function followed by **gc_DropCall( )** and **gc_ReleaseCall( )** functions. See the *GlobalCall Country Dependent Parameters (CDP) Reference* for protocol specific limitations. However, anytime a GCEV_DISCONNECTED event is received in the Accepted state, the **gc_DropCall( )** function can be issued.

Different technologies and protocols support some or all of the cause values defined above; refer to the appropriate *GlobalCall Technology User's Guide* for valid causes for your specific technology.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

/*
 * Assume the following has been done:
 *    1. Opened line devices for each time slot on DTIB1.
 *    2. Wait for a call using gc_WaitCall()
 *    3. The application has chosen to terminate the call
 *                  OR
 *       the unsolicited event GCEV_DISCONNECTED has arrived
 * Note: A call may be dropped from any state other than IDLE or NULL
 */
int drop_call(CRN crn)
{
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */

   if (gc_DropCall(crn, GC_NORMAL_CLEARING, EV_ASYNC) != GC_SUCCESS) {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               metaevent.evtdev, gc_error, msg);
      return(gc_error);
   }
   /*
    * gc_DropCall() is terminated by the GCEV_DROPCALL event.
    * Application must then release the call using gc_ReleaseCall().
    */
   return (0);
}
```

Errors

If this function returns a <0 to indicate failure or if the GCEV_TASKFAIL event is received, use **gc_ErrorValue( )** or **gc_ResultValue( )**, respectively, and the **gc_ResultMsg( )** function as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

See Also

- **gc_MakeCall( )**
- **gc_ReleaseCall( )**
- **gc_WaitCall( )**

Name: int gc_ErrorValue(gc_errorp, cclibidp, cclib_errorp)

Inputs:
- int *gc_errorp — • location to store GlobalCall error
- int *cclibidp — • location to store call control library ID
- long *cclib_errorp — • location to store call control library error description

Returns: 0 if error value successfully retrieved
<0 if fails to retrieve error value

Includes: gclib.h
gcerr.h

Category: system control and tools

Mode: synchronous

Technology: ■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ■ Analog

■ Description

The **gc_ErrorValue( )** function gets an error value/failure reason code associated with the last GlobalCall function call. To retrieve an error, this function must be called immediately after a GlobalCall function failed. This function returns the GlobalCall error code, **gc_errorp*, as well as the lower level error code associated directly with the call control library, **cclib_errorp*. The GlobalCall error code is a generic error that has a consistent meaning across all call control libraries.

A call control library error may be more specific to the supported technology. These error values provide optimal debugging and troubleshooting for the application developer. For example, a time-out error may occur for multiple reasons when establishing a call. The specific reasons may vary for different network interfaces (ISDN time-out errors differ from those in an R2 MFC protocol). Each of these call control library time-out errors are mapped to *EGC_TIMEOUT*. However, the specific time-out error detected by the call control library will be available through *cclib_errorp*.

Parameter	Description
gc_errorp:	pointer to the location where the GlobalCall error code will be stored
cclibidp:	pointer to the location to store the identification number of

Parameter	Description
	the call control library where the error occurred
cclib_errorp:	pointer to the location to store the call control library error description that is uniquely associated to its own library

Termination Event: None.

■ Cautions

To aid in debugging, both the **gc_errorp** and the **cclib_errorp** values should be retrieved.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

void print_error_values(void)
{
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */
   char           *lib_name;        /* library name for cclibid */

   /* This could be called when any function fails;
    * to print the error values */

   if (gc_ErrorValue( &gc_error, &cclibid, &cc_error) == GC_SUCCESS) {
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf("GlobalCall error 0x%lx - %s\n", gc_error, msg);
      gc_ResultMsg( cclibid, cc_error, &msg);
      gc_CCLibIDToName(cclibid, &lib_name);
      printf("% library had error 0x%lx - %s\n", lib_name, cc_error, msg);
   } else {
      printf("Could not get error value\n");
   }
}
```

■ Errors

If this function returns a <0 to indicate failure, then at least one of its input parameters is NULL.

See Also

- **gc_ResultMsg( )**

Name:	int gc_GetANI(crn, ani_buf)	
Inputs:	CRN crn	• call reference number
	char *ani_buf	• buffer for storing ANI digits
Returns:	0 if successful	
	<0 if failure	
Includes:	gclib.h	
	gcerr.h	
Category:	optional feature	
Mode:	synchronous	
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit	
	■ Analog	

■ Description

The **gc_GetANI( )** function returns ANI information received during call establishment/setup. If the ANI information is not available, an error will be sent and the **gc_GetANI( )** function fails.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
crn:	Call Reference Number
ani_buf:	address of the buffer where ANI is to be loaded. The returned digits will be terminated with '\0'.

Termination Event: None.

■ Cautions

The **ani_buf** buffer MUST BE large enough to store the largest expected ANI string length (including the zero terminator), which is defined by GC_ADDRSIZE.

If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

/*
 * Assume the following has been done:
 *    1. Opened line devices for each time slot on DTIB1.
 *    2. Wait for a call using gc_WaitCall()
 *    3. An event has arrived and has been converted to a metaevent
 *       using gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows NT)
 *    4. The event is determined to be a GCEV_OFFERED event
 */
int get_ani(void)
{
   CRN           crn;                /* call reference number */
   char          ani_buf[GC_ADDRSIZE]; /* Buffer for ANI digits */
   int           cclibid;            /* cclib id for gc_ErrorValue() */
   int           gc_error;           /* GlobalCall error code */
   long          cc_error;           /* Call Control Library error code */
   char          *msg;               /* points to the error message string */

   /*
    * Get the calling party number
    */
   crn = metaevent.crn;
   if (gc_GetANI(crn, ani_buf) != GC_SUCCESS) {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               metaevent.evtdev, gc_error, msg);
      return(gc_error);
   }
   /* Application can answer, accept, or terminate the call at this
    * point, based on the ANI information. */
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

See Also

- **gc_ReqANI( )**
- **gc_WaitCall( )**

Name:	int gc_GetBilling(crn, billing_buf)	
Inputs:	CRN crn	• call reference number
	char *billing_buf	• buffer for billing information
Returns:	0 if successful	
	<0 if failure	
Includes:	gclib.h	
	gcerr.h	
Category:	optional feature	
Mode:	synchronous	
Technology:	■ ISDN PRI ❑ E-1 CAS ❑ T-1 robbed bit	
	❑ Analog	

■ Description

The **gc_GetBilling( )** function gets the charge information associated with the specified call. The charge information is in ASCII string format**.** The information is retrieved from the GlobalCall software.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
crn:	Call Reference Number
billing_buf:	address of the buffer where the requested information is stored. Refer to the appropriate *GlobalCall Technology User's Guide* for the format and usage of this field.

Termination Event: None.

■ Cautions

Ensure that the **billing_buf** buffer is large enough to store the greatest expected amount of billing information, which is defined by GC_BILLSIZE.

If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

■ Example

```
/*
 * Assume the following has been done:
 * 1. device was opened (e.g. :N_dtiB1T1:P_isdn, :N_dtiB1T2:P_isdn, etc...)
 * 2. gc_WaitCall() has been issued to wait for a call.
 * 3. gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows NT) has been
 *    called to convert the event into metaevent.
 * 4. a GCEV_OFFERED has been detected.
 * 5. a call has been connected.
 * 6. the call has been disconnected after conversation.
 */


#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>

/* This is only available for AT&T 4ESS switch. */

int get_billing_info(CRN crn, char *billing_buffer)
{
  LINEDEV  ddd;             /* Line device */
  int      gc_err;          /* GlobalCall Error Code */
  int      cclibid;         /* Call Control library ID */
  long     cclib_err;       /* Call Control Error Code */
  char     *msg;            /* Error Message */

  if(gc_CRN2LineDev(crn, &ddd) != GC_SUCCESS) {
    gc_ErrorValue(&gc_err, &cclibid, &cclib_err);
    gc_ResultMsg(cclibid, cclib_err, &msg);
    printf ("Error: gc_CRN2LineDev ErrorValue: %d - %s\n",
             cclib_err, msg);
    return(cclib_err);
  }

  if(gc_GetBilling(crn, billing_buffer) != GC_SUCCESS) {
    gc_ErrorValue(&gc_err, &cclibid, &cclib_err);
    gc_ResultMsg(cclibid, cclib_err, &msg);
    printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
             ddd, gc_err, msg);
    return(cclib_err);
  }

  return(0);
}
```

Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

See Also

- None

Name:	int gc_GetCallInfo(crn, info_id, valuep)	
Inputs:	CRN crn	• call reference number
	int info_id	• call info ID
	char *valuep	• pointer to info buffer
Returns:	0 if successful	
	<0 if failure	
Includes:	gclib.h	
	gcerr.h	
	gcisdn.h (for applications that use ISDN symbols)	
Category:	interface specific	
Mode:	synchronous	
Technology:	■ ISDN PRI ■ E-1 CAS ❑ T-1 robbed bit	
	■ Analog	

■ Description

The **gc_GetCallInfo( )** function gets information for the call. You can use this function at any time. The application can retrieve only one type of information at a time.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
crn:	Call Reference Number
info_id:	identifies parameter requested, see *Table 29* for definitions
valuep:	buffer address where the requested information is stored

Table 29. GetCallInfo( ) info_id Parameter ID Definitions

info_id Parameter	Definition	Technology	*valuepFormat
CALLED_SUBS	Called party subaddress	ISDN	string
CALLNAME	Calling party's name	Analog	string
CALLTIME	Time and date call was made	Analog	string
CATEGORY_DIGIT	Category digit	E-1 CAS	character
CONNECT_TYPE	Defines the type of connection returned by call progress analysis	Analog	character
U_IES	Unformatted user-to-user Information Elements	ISDN	string
UUI	User-to-User Information	ISDN	string

Termination Event: None.

■ Cautions

- An incoming Information Element (IE) is not accepted until the existing IE is read by the application. A GCEV_NOUSRINFOBUF event is sent to the application.
- **Multiple IEs in the same message**: Only happens to Network Specific Facility IE. When it happens, the library stores all IEs. If the combination of IEs is longer than the storage capacity, the Library discards the overflow IEs and issues a GCEV_NOFACILITYBUF event to the application.

- Ensure that the application verifies that the buffer pointed to by the **valuep** parameter is large enough to hold the information requested by the **info_id** parameter.

If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

■ Example

```
/*
 * Assume the following has been done:
 * 1. device has been opened (e.g. :N_dtiB1T1:P_isdn,
 *    :N_dtiB1T2:P_isdn, etc...)
 * 2. gc_WaitCall() has been issued to wait for a call.
 * 3. gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows NT) has been
 *    called to convert the event into metaevent.
 * 4. a GCEV_OFFERED has been detected.
 */


#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>

/*
 * the variable info_id parameter(s) defines the information
 * requested from the network.
 * The variable valuep stores the returned information.
 */

int get_call_info(CRN crn, int info_id, char *valuep)
{
  LINEDEV  ddd;             /* Line device */
  int      gc_err;          /* GlobalCall Error Value */
  int      cclibid;         /* Call Control library ID */
  long     cclib_err;       /* Call Control Error Value */
  char     *msg;            /* Error Message */

  if(gc_CRN2LineDev(crn, &ddd) != GC_SUCCESS) {
    gc_ErrorValue(&gc_err, &cclibid, &cclib_err);
    gc_ResultMsg(cclibid, cclib_err, &msg);
    printf ("Error: gc_CRN2LineDev ErrorValue: %d - %s\n",
             cclib_err, msg);
    return(cclib_err);
  }

  if(gc_GetCallInfo(crn, info_id, valuep) != GC_SUCCESS) {
    gc_ErrorValue(&gc_err, &cclibid, &cclib_err);
    gc_ResultMsg(cclibid, cclib_err, &msg);
    printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
             ddd, gc_err, msg);
    return(cclib_err);
  }
```

```
  return(0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- None

Name:	int gc_GetCallState(crn, state_buf)
Inputs:	CRN crn • call reference number int *state_buf • pointer to variable for returning call state
Returns:	0 if successful <0 if failure
Includes:	gclib.h gcerr.h
Category:	system control and tools
Mode:	synchronous
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ■ Analog

■ Description

The **gc_GetCallState( )** function acquires the state of the call associated with the CRN. The acquired state will be associated with the last message received by the application. This function is especially useful when an error occurs and the application requires an update as to whether the call state has changed.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description	
crn:	Call Reference Number	
state_buf:	pointer to the location where the call state value will be returned. Possible state values are:	
	GCST_NULL	call released
	GCST_OFFERED	inbound call received
	GCST_ACCEPTED	call accepted
	GCST_CONNECTED	call connected
	GCST_DIALING	outbound call request
	GCST_ALERTING	call alerted sent or received
	GCST_DISCONNECTED	call disconnected from network
	GCST_IDLE	call is not active
	State transition diagrams and call state definitions are presented in *paragraph 3.3. GlobalCall Call States*.	

Termination Event: None.

■ Cautions

Due to the process latency time, the state value acquired through the **gc_GetCallState( )** function may lag behind the current call state in the protocol state machine. If the two state values differ, the acquired state value is always behind the actual state. This is especially evident in the process of establishing an outbound call. The state acquired by the application will be associated with the latest event received by the application.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

#define MAXCHAN   30               /* max. number of channels in system */
/*
 * Data structure which stores all information for each line
 */
struct linebag {
   LINEDEV  ldev;                  /* GlobalCall line device handle */
   CRN      crn;                   /* GlobalCall API call handle */
   int      state;                 /* state of first layer state machine */
} port[MAXCHAN+1];
```

```
struct linebag *pline;              /* pointer to access line device */

int get_call_state(int port_num)
{
   LINEDEV        ldev;             /* line device ID */
   CRN            crn;              /* call reference number */
   int            call_state;       /* current state of call */
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */

   /* Find info for this time slot, specified by 'port_num' */
   /* (Assumes port_num is valid) */
   pline = port + port_num;
   crn = pline -> crn;
   /*
    * Retrieve the call state and save it.
    */
   if (crn) {
      if (gc_GetCallState( crn, &call_state) != GC_SUCCESS) {
         /* process error return as shown */
         gc_ErrorValue( &gc_error, &cclibid, &cc_error);
         gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
         if (gc_CRN2LineDev( crn, &ldev) != GC_SUCCESS) {
            /* get and process error */
         }
         printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
                  ldev, gc_error, msg);
         return(gc_error);
      }
   }

   pline->state = call_state;
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- None

Name: int gc_GetCRN(crnp, metaeventp)

Inputs: CRN *crnp • pointer to returned CRN
METAEVENT *metaeventp • pointer to a metaevent block

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: system control and tools

Mode: synchronous

Technology: ■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit
■ Analog

■ Description

The **gc_GetCRN( )** function gets the CRN for the event to which the pointer **metaeventp** is pointing. This **metaeventp** pointer is filled in by the **gc_GetMetaEvent( )** function or the **gc_GetMetaEventEx( )** function (Windows NT extended asynchronous mode only).

The application can access the CRN directly from the metaevent using the **crn** field of **metaeventp** rather than using this **gc_GetCRN( )** function. The **gc_GetCRN( )** function is supported for backward compatibility but is not otherwise needed since the CRN is available when the metaevent is returned from the **gc_GetMetaEvent( )** function or the **gc_GetMetaEventEx( )** function (Windows NT extended asynchronous mode only).

If the event is call related, the **metaeventp crn** field contains the CRN. After a call to the **gc_GetCRN( )** function, the ***crnp** also contains the CRN. If the event is not call related but rather associated with the line device, the **metaeventp crn** field is set to 0. In this case, after a call to the **gc_GetCRN( )** function, the ***crnp** is also 0. The line device may also be obtained directly from the metaevent via the **metaeventp linedev** field.

Parameter	Description
crnp:	pointer to the memory address where the call reference number is stored.
metaeventp:	pointer to the METAEVENT structure filled in by

Parameter	Description
	gc_GetMetaEvent( ) or the **gc_GetMetaEventEx( )** function (Windows NT extended asynchronous mode only).

Termination Event: None.

■ Cautions

None

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

/*
 * Assume the following has already been done:
 *    1. Opened line devices for each time slot on DTIB1.
 *    2. Wait for a call using gc_WaitCall()
 *    3. An event has arrived and has been converted to a metaevent
 *       using gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows NT)
 */
CRN get_crn(METAEVENT *metaeventp)
{
   CRN           crn;             /* call reference number */
   int           cclibid;         /* cclib id for gc_ErrorValue() */
   int           gc_error;        /* GlobalCall error code */
   long          cc_error;        /* Call Control Library error code */
   char          *msg;            /* points to the error message string */

   if (gc_GetCRN(&crn, metaeventp) != GC_SUCCESS) {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               metaevent.evtdev, gc_error, msg);
      return(0);
   }
   else {
      /*
       * Else return the CRN and next issue the GlobalCall function call
       * using the CRN.
       */
      return(crn);
   }
}
```

Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

See Also

- **gc_GetLineDev( )**
- **gc_GetMetaEvent( )**
- **gc_GetMetaEventEx( )** (Windows NT extended asynchronous mode only)
- **gc_MakeCall( )**
- **gc_WaitCall( )**

Name:	int gc_GetDNIS(crn, dnis_buf)	
Inputs:	CRN crn	• call reference number
	char *dnis_buf	• buffer to store DNIS info
Returns:	0 if successful	
	<0 if failure	
Includes:	gclib.h	
	gcerr.h	
Category:	optional feature	
Mode:	synchronous	
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit	
	❑ Analog	

■ Description

The **gc_GetDNIS( )** function gets the DNIS information (DDI digits) associated with a specific CRN. The DDI digits are in ASCII string format and ends with '\0'.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
crn:	Call Reference Number
dnis_buf:	address of the buffer where the DNIS is stored.

Termination Event: None.

■ Cautions

The **dnis_buf** buffer MUST BE large enough to store the largest expected DNIS string length, which is defined by GC_ADDRSIZE.

If the application needs more DDI digits, the application can use the **gc_CallAck( )** function to request more digits, if the protocol supports this feature. The **gc_GetDNIS( )** function may be called again to retrieve these digits.

If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <string.h>
#include <gclib.h>
#include <gcerr.h>

/*
 * Assume the following has been done:
 *    1. 'maxddi' has been setup depending on needs
 *       of application/protocol.
 *    2. Line devices have been opened for each time slot on dtiB1.
 *    3. Wait for a call using gc_WaitCall()
 *    4. An event has arrived and has been converted to a metaevent
 *       using gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows NT)
 *    5. The event is determined to be a GCEV_OFFERED event
 */
int get_dnis(void)
{
   CRN           crn;                  /* call reference number */
   int           maxddi = 10;          /* maximum allowable DDI digits */
   char          dnis_buf[GC_ADDRSIZE]; /* DNIS digit Buffer */
   int           cclibid;              /* cclib id for gc_ErrorValue() */
   int           gc_error;             /* GlobalCall error code */
   long          cc_error;             /* Call Control Library error code */
   char          *msg;                 /* points to the error message string */

   /* 1st get the crn */
   if (gc_GetCRN(&crn, &metaevent) != GC_SUCCESS) {
      /* handle the gc_GetCRN error */
   }

   /*
    * Get called party number and check that there were not too
    * many digits collected.
    */
   if (gc_GetDNIS(crn, dnis_buf) != GC_SUCCESS) {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               metaevent.evtdev, gc_error, msg);
      return(gc_error);
   }

   if (strlen(dnis_buf) <= maxddi) {
      /*
       * Process called party number as needed by the application.
       */
   } else {
      /*
      * Drop the call if number of DDI digits exceeds maximum limit
       */
      if (gc_DropCall(crn, GC_NORMAL_CLEARING, EV_ASYNC) != GC_SUCCESS) {
         /* process error return from gc_DropCall()  */
```

```
      }
   }
   /*
    * Application can answer, accept, or terminate the call at this
    * point, based on the DNIS information.
    */
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_CallAck( )**
- **gc_GetANI( )**
- **gc_WaitCall( )**

Name: int gc_GetLineDev(linedevp, metaeventp)

Inputs: LINEDEV *linedevp • pointer to returned line device
METAEVENT *metaeventp • pointer to metaevent block

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: system control and tools

Mode: synchronous

Technology: ■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ■ Analog

■ Description

The **gc_GetLineDev( )** function gets a line device associated with the event received from the event queue. If this function is called for an event that is not a GlobalCall event, then the ***linedevp** parameter is set to 0. The line device may also be retrieved using the linedev field in the METAEVENT structure instead of using this function.

The **gc_GetLineDev( )** function is supported for backward compatibility but is not otherwise needed since the line device ID is available when the metaevent is returned from the **gc_GetMetaEvent( )** function or the **gc_GetMetaEventEx( )** function (Windows NT extended asynchronous mode only).

Parameter	Description
linedevp:	pointer to the location where the output LINEDEV is stored
metaeventp:	pointer to the METAEVENT structure filled in by **gc_GetMetaEvent( )** or the **gc_GetMetaEventEx( )** function (Windows NT extended asynchronous mode only)

Termination Event: None.

■ Cautions

None

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

/*
 * Assume the following has been done:
 *    1. Opened line devices for each time slot on DTIB1.
 *    2. Wait for a call using gc_WaitCall()
 *    3. An event has arrived and has been converted to a metaevent
 *       using gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows NT)
 *    4. The event is determined to be a GCEV_OFFERED event
 */
int get_linedev(LINEDEV *ldevp)
{
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */

   /*
    * Get Line Device corresponding to this event
    */
   if (gc_GetLineDev(ldevp, &metaevent) != GC_SUCCESS) {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               metaevent.evtdev, gc_error, msg);
      return(gc_error);
      }

   /*
    * The line device ID may then be used for functions like
    * gc_SetParm(), gc_SetUsrAttr(), and gc_GetVoiceH().
    */
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

See Also

- **gc_GetCRN( )**
- **gc_GetMetaEvent( )**
- **gc_GetMetaEventEx( )** (Windows NT extended asynchronous mode only)

Name:	int gc_GetLinedevState(linedev, type, statebuf)	
Inputs:	LINEDEV linedev	• line device
	int type	• specifies type of line device
	int *statebuf	• pointer to location of line device state status
Returns:	0 if successful	
	<0 if failure	
Includes:	gclib.h	
	gcerr.h	
	gcisdn.h	
Category:	optional feature	
Mode:	synchronous	
Technology:	■ ISDN PRI ❑ E-1 CAS ❑ T-1 robbed bit	
	❑ Analog	

■ Description

The **gc_GetLinedevState( )** function retrieves status of the line device specified by the **linedev** parameter.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
linedev:	GlobalCall line device
type:	specifies B channel or D channel device type associated with **linedev**, valid values are: • GCGLS_BCHANNEL get state of B channel • GCGLS_DCHANNEL get state of D channel

Parameter	Description
statebuf:	location to which state information is written, valid state values are: • for B channel: • GCLS_INSERVICE — B channel is in service • GCLS_MAINTENANCE — B channel is in maintenance • GCLS_OUT_OF_SERVICE — B channel is out of service • for D channel: • DATA_LINK_UP — layer 2 operable • DATA_LINK_DOWN — layer 2 inoperable

Termination Event: None.

■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>

int get_line_dev_state(void)
{
  LINEDEV  bdd;             /* Board level device */
  LINEDEV  ddd;             /* Line device */
  char     bdevname[40];    /* Board device name */
  char     ldevname[40];    /* Line device name */
  int      type;            /* Type of line device */
  int      statebuf;        /* Buffer to store line device state */
  int      gc_err;          /* GlobalCall Error Code */
```

```
  int      cclibid;         /* Call Control library ID */
  long     cclib_err;       /* Call Control Error Code */
  char     *msg;            /* Error Message */

  sprintf(bdevname, "dtiB1");
  sprintf(ldevname, "dtiB1T1");

  /*
   * Open two devices, dtiB1 and dtiB1T1.
   */
  if(gc_Open(&bdd, bdevname, 0) != GC_SUCCESS) {
    gc_ErrorValue(&gc_err, &cclibid, &cclib_err);
    gc_ResultMsg(cclibid, cclib_err, &msg);
    printf ("Error:gc_Open, ErrorValue: %d - %s\n", gc_err, msg);
    return(cclib_err);
  }

  if(gc_Open(&ddd, ldevname, 0) != GC_SUCCESS) {
    gc_ErrorValue(&gc_err, &cclibid, &cclib_err);
    gc_ResultMsg(cclibid, cclib_err, &msg);
    printf ("Error:gc_Open, ErrorValue: %d - %s\n", gc_err, msg);
    return(cclib_err);
  }

  /*
   * Find the status of the board.
   * the D Channel can be in one of two states DATA_LINK_UP or
   * DATA_LINK_DOWN.
   */
  type = GCGLS_DCHANNEL;

  if(gc_GetLinedevState(bdd, type, &statebuf) != GC_SUCCESS) {
    gc_ErrorValue(&gc_err, &cclibid, &cclib_err);
    gc_ResultMsg(cclibid, cclib_err, &msg);
    printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
             bdd, gc_err, msg);
    return(cclib_err);
  }
  else {
    printf("D Channel Status: %s\n", statebuf);
  }

  /*
   * Find the status of the line.
   * the B Channel can be in one of three states GCLS_INSERVICE,
   * GCLS_MAINTENANCE, or GCLS_OUT_OF_SERVICE.
   */
  type = GCGLS_BCHANNEL;

  if(gc_GetLinedevState(ddd, type, &statebuf) != GC_SUCCESS) {
    gc_ErrorValue(&gc_err, &cclibid, &cclib_err);
    gc_ResultMsg(cclibid, cclib_err, &msg);
    printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
             ddd, gc_err, msg);
    return(cclib_err);
  }
  else {
    printf("B Channel Status: %s\n", statebuf);
  }

  gc_Close(bdd);
  gc_Close(ddd);

  return (0);
}
```

Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

See Also

- **gc_SetChanState( )**

Name:	int gc_GetMetaEvent(metaeventp)
Inputs:	METAEVENT *metaeventp • pointer to METAEVENT block
Returns:	0 if successful <0 if failure
Includes:	gclib.h gcerr.h
Category:	system control and tools
Mode:	synchronous
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ■ Analog

■ Description

The **gc_GetMetaEvent( )** function maps the current SRL event into a metaevent that stores the GlobalCall and non-GlobalCall event information. This function returns an event to the UNIX or Windows NT application in the form of a metaevent. This metaevent is a data structure that explicitly contains the information describing the event. This data structure provides uniform information retrieval among call control libraries and across operating systems; see *paragraph 5.4. METAEVENT* for data structure details.

The current SRL event information is not changed or altered by calling the **gc_GetMetaEvent( )** function to retrieve event information. This function may be used as a convenience function to retrieve the event information for all SRL events. Whether the event is a GlobalCall event or any other SRL event, the SRL event information (e.g., evtdatap, evttype, etc.) may be retrieved from the METAEVENT data structure instead of using SRL functions to retrieve this information, see *paragraph 5.4. METAEVENT* for data structure details. If the metaevent is a GlobalCall event, the GCME_GC_EVENT bit in the metaevent flag field will be set. When this bit is set, the GlobalCall related fields in the METAEVENT data structure will be filled with GlobalCall event information.

Parameter	Description
metaeventp:	pointer to the structure of metaevent data filled in by this function; see *paragraph 5.4. METAEVENT* for data structure details.

Termination Event: None.

■ Cautions

The **gc_GetMetaEvent( )** [or **gc_GetMetaEventEx( )** (Windows NT only)] function MUST BE the first function called before processing any GlobalCall event.

For Windows NT applications, when using the extended asynchronous mode, the **gc_GetMetaEventEx( )** function must be the first function called before processing any GlobalCall event. For all other Windows NT modes, use the **gc_GetMetaEvent( )** function.

The **gc_GetMetaEvent( )** and **gc_GetMetaEventEx( )** functions should not be used in the same application.

■ Example

The following code illustrates calling the **gc_GetMetaEvent( )** function in response to receiving an event via the SRL.

```
if(sr_waitevt(timeout) != -1) {     /* i.e. an event occurred */
   retcode = gc_GetMetaEvent(&metaevent);
   if (retcode <0) {
      /* get and process the error */
   } else {
      /* Continue with normal processing */
   }
}
```

OR

```
handler(...)
{
   retcode = gc_GetMetaEvent(&metaevent);
   if (retcode <0 ) {
      /* get and process the error */
   } else {
      /* Continue with normal processing */
  }
}
```

To retrieve and process information associated with an event, the following example code can be used. This code returns the event type, event data pointer, event length and event device associated with the event from either the handler or after a **sr_waitevt( )** function call.

```
   retcode = gc_GetMetaEvent(&metaevent);
   if (retcode <0) {
          /* get and process error */
   } else {
         /* Can now access SRL information for any GlobalCall or
            non-GlobalCall event using: */
         /* metaevent.evtdatap */
         /* metaevent.evtlen */
         /* metaevent.evtdev */
         /* metaevent.evttype */
       if (metaevent.flags & GCME_GC_EVENT) {
            /* process GlobalCall event here */
       } else {
            /* process non-GlobalCall event here */
       }
   }
```

An alternative for determining whether an event is a GlobalCall API event or a non-GlobalCall event is as follows:

```
   evttype = sr_getevttype();
   if ((evttype & DT_GC) == DT_GC) {
         /* process GlobalCall event */
   } else {
         /* process non-GlobalCall event */
   }
```

The following code illustrates retrieving event information from the METAEVENT structure while making a call:

```
#include <windows.h>             /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#define MAXCHAN   30                /* max. number of channels in system */
#define NULL_STATE 0
#define DIALING_STATE   1
#define ALERTING_STATE  2
#define CONNECTED_STATE 3
/*
 * Data structure which stores all information for each line
 */
struct linebag {
   LINEDEV  ldev;                   /* network line device handle */
   CRN      crn;                    /* GlobalCall API call handle */
   int      state;                  /* state of first layer state machine */
} port[MAXCHAN+1];

struct linebag *pline;              /* pointer to access line device */

/*
 * Assume the following has been done:
 *    1. Opened line devices for each time slot on DTIB1.
 *    2. Application is in the NULL state
 * Examples are given in ASYNC mode
 * Error handling is not shown
 */
int makecall(int port_num, char *numberstr)
{
```

```
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */
   long           evttype;          /* type of event */

   /* Find info for this time slot, specified by 'port_num' */
   /* (Assumes port_num is valid) */
   pline = port + port_num;

   if (gc_MakeCall(pline -> ldev, &pline -> crn, numberstr, NULL, 0, EV_ASYNC) !=
       GC_SUCCESS) {
      /* process error and return */
   }

   pline -> state = DIALING_STATE;

   for (;;) {
      sr_waitevt(-1L);              /* wait forever */

      /* Get the next event */
      if (gc_GetMetaEvent(&metaevent) != GC_SUCCESS) {
         /* process error */
      }

      evttype = metaevent.evttype;
      if (metaevent.flags & GCME_GC_EVENT) {
         /* process GlobalCall event */
         switch (pline -> state) {
            case DIALING_STATE:
               switch (evttype) {
                  case GCEV_ALERTING:
                     pline -> state = ALERTING_STATE;
                     break;
                  case GCEV_CONNECTED:
                     pline -> state = CONNECTED_STATE;
                     /*
                      * Can now do voice functions, etc.
                      */
                     return(0);                 /* SUCCESS RETURN POINT */
                  default:
                     /* handle other events here */
                     break;
               }
               break;

            case ALERTING_STATE:
               switch (evttype) {
                  case GCEV_CONNECTED:
                     pline -> state = CONNECTED_STATE;
                     /*
                      * Can now do voice functions, etc.
                      */
                     return(0);                 /* SUCCESS RETURN POINT */
                  default:
                     /* handle other events here */
                     break;
               }
               break;
         }
      } else {
         /* Process non-GlobalCall event */
      }
   }
}
```

The following code illustrates retrieving event information from the METAEVENT structure while waiting for a call:

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

#define MAXCHAN   30               /* max. number of channels in system */
#define NULL_STATE 0
#define CONNECTED_STATE 3
#define OFFERED_STATE   4
#define ACCEPTED_STATE  5

/*
 * Data structure which stores all information for each line
 */
struct linebag {
   LINEDEV  ldev;                  /* network line device handle */
   CRN      crn;                   /* GlobalCall API call handle */
   int      state;                 /* state of first layer state machine */
} port[MAXCHAN+1];

struct linebag *pline;             /* pointer to access line device */

/*
 * Assume the following has been done:
 *    1. Opened line devices for each time slot on DTIB1.
 *    2. Application is in the NULL state
 *    3. A gc_WaitCall() has been successfully issued
 *
 * Examples are given in ASYNC mode
 * Error handling is not shown
 *
 */
int waitcall(int port_num)
{
   int            cclibid;         /* cclib id for gc_ErrorValue() */
   int            gc_error;        /* GlobalCall error code */
   long           cc_error;        /* Call Control Library error code */
   char           *msg;            /* points to the error message string */

   long           evttype;         /* type of event */

   /* Find info for this time slot, specified by 'port_num' */
   /* (Assumes port_num is valid) */
   pline = port + port_num;

   for (;;) {
      sr_waitevt(-1L);             /* wait forever */

      /* Get the next event */
      if (gc_GetMetaEvent(&metaevent) != GC_SUCCESS) {
         /* process error return */
      }

      evttype = metaevent.evttype;
      if (metaevent.flags & GCME_GC_EVENT) {
         /* process GlobalCall event */
         switch (pline -> state) {
            case NULL_STATE:
               switch (evttype) {
                  case GCEV_OFFERED:
```

```
                  pline -> state = OFFERED_STATE;
                  accept_call();
                  break;
               default:
                  /* handle other events here */
                  break;
            }
            break;

         case OFFERED_STATE:
            switch (evttype) {
               case GCEV_ACCEPT:
                  pline -> state = ACCEPTED_STATE;
                  answer_call();
                  break;
               default:
                  /* handle other events here */
                  break;
            }
            break;
         case ACCEPTED_STATE:
            switch (evttype) {
               case GCEV_ANSWERED:
                  pline -> state = CONNECTED_STATE;
                  /*
                   * Can now do voice functions, etc.
                   */
                  return(0);                /* SUCCESS RETURN POINT */
               default:
                  /* handle other events here */
                  break;
            }
            break;
      }
   } else {
      /* Process non-GlobalCall event */
   }
  }
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_GetCRN( )**
- **gc_GetLineDev( )**
- **gc_GetMetaEventEx( )** (Windows NT extended asynchronous mode only)
- **gc_ResultValue( )**

Name:	int gc_GetMetaEventEx(metaeventp, evt_handle) [Windows NT extended asynchronous mode only]	
Inputs:	unsigned long evt_handle	• SRL event handle
	METAEVENT *metaeventp	• pointer to METAEVENT block
Returns:	0 if successful <0 if failure	
Includes:	windows.h gclib.h gcerr.h	
Category:	system control and tools	
Mode:	synchronous	
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ■ Analog	

■ Description

The **gc_GetMetaEventEx( )** function maps the current SRL event into a metaevent that passes the SRL event handle to the application or thread and stores the GlobalCall and non-GlobalCall event information in the METAEVENT data structure. This function returns an event to the Windows NT application running multithreads in the extended asynchronous mode in the form of a metaevent and an SRL event handle. This metaevent is a data structure that explicitly contains the information describing the event. This data structure provides uniform information retrieval among call control libraries and across operating systems; see *paragraph 5.4. METAEVENT* for data structure details. The event handle is passed to the application by the SRL:

- when the **sr_waitevtEx( )** function is called, typically for multithreaded applications or
- when the SRL handlers are called; the event handle is the first parameter in the function call.

The current SRL event information is not changed or altered by calling the **gc_GetMetaEventEx( )** function to retrieve event information. This function may be used as a convenience function to retrieve the event information for all

SRL events. Whether the event is a GlobalCall event or any other SRL event, the SRL event information (e.g., evtdatap, evttype, etc.) may be retrieved from the METAEVENT data structure instead of using SRL functions to retrieve this information, see *paragraph 5.4. METAEVENT* for data structure details. If the metaevent is a GlobalCall event, the GCME_GC_EVENT bit in the metaevent flag field will be set. When this bit is set, the GlobalCall related fields in the METAEVENT data structure will be filled with GlobalCall event information.

Parameter	Description
evt_handle:	SRL event handle used to identify event with a particular thread.
metaeventp:	pointer to the structure of metaevent data filled in by this function; see *paragraph 5.4. METAEVENT* for data structure details.

Termination Event: None.

■ Cautions

The **gc_GetMetaEvent( )** or **gc_GetMetaEventEx( )** function MUST BE the first function called before processing any GlobalCall event

When using the extended asynchronous mode, the **gc_GetMetaEventEx( )** function must be the first function called before processing any GlobalCall event. For all other Windows NT modes, use the **gc_GetMetaEvent( )** function.

The **gc_GetMetaEvent( )** and **gc_GetMetaEventEx( )** functions should not be used in the same application.

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.

■ Example

The following code illustrates event retrieval wherein the SRL gets the event and then the extended **gc_GetMetaEventEx( )** function fills in the METAEVENT data structure.

```
/*
 * Do SRL event processing
 */
hdlcnt = 0;
hdls[ hdlcnt++ ] = GetGlobalCallHandle();
hdls[ hdlcnt++ ] = GetVoiceHandle();

/* Wait selectively for devices that belong to this thread */

rc = sr_waitevtEx(   hdls,
                     hdlcnt,
                     PollTimeout_ms,
                     &evtHdl
                 );

if (rc != SR_TMOUT) {
      /*
       *  Update
       */
       rc = gc_GetMetaEventEx(&g_Metaevent, evtHdl);
       if (rc != GC_SUCCESS) {
                CheckError(rc,"gc_GetMetaEventEx");
                return -1;
       }

          rc = vProcessCallEvents(  );
       }
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_GetCRN( )**
- **gc_GetLineDev( )**
- **gc_GetMetaEvent( )**
- **gc_ResultValue( )**

Name:	int gc_GetNetworkH(linedev, networkhp)	
Inputs:	LINEDEV linedev	• GlobalCall line device handle
	int *networkhp	• pointer to returned network device handle
Returns:	0 if successful <0 if failure	
Includes:	gclib.h gcerr.h	
Category:	system control and tools	
Mode:	synchronous	
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ■ Analog	

■ Description

The **gc_GetNetworkH( )** function returns the network device handle associated with the specified line device **linedev**. The ***networkhp** parameter is actually the SRL device handle of the network resource associated with the line device. The ***networkhp** parameter can be used as an input to functions requiring a network handle, such as the SCbus routing function **nr_scroute( )**.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
linedev:	GlobalCall line device handle
networkhp:	address at which the network device handle is to be stored.

Termination Event: None.

■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

/*
 * Assume the following has been done:
 *    1. A line device (ldev) has been opened, specifying a
 *          network time slot and a protocol
 *          For example, 'devicename' could be ":N_dtiB1T1:P_br_r2_i".
 */
int route_fax_to_gc(LINEDEV ldev, int faxh)
{
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */
   int            networkh;         /* network device handle */

   if (gc_GetNetworkH(ldev, &networkh) == GC_SUCCESS) {
      /*
       * Route the fax resource to the network device in
       * a full duplex manner.
       */
      if (nr_scroute(networkh,SC_DTI,faxh,SC_FAX,SC_FULLDUP) == -1) {
         /* process error */
      }
      else {
         /* proceed with the fax call */
      }


   } else {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               ldev, gc_error, msg);
      return(gc_error);
   }
   /*
    * Application may now generate or wait for a call on this line
    * device.
    */
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

See Also

- **gc_GetVoiceH( )**

Name:	int gc_GetParm(linedev, parm_id, valuep)
Inputs:	LINEDEV linedev • GlobalCall line device handle int parm_id • parameter ID GC_PARM *valuep • pointer to buffer where value will be stored
Returns:	0 if successful <0 if failure
Includes:	gclib.h gcerr.h gcisdn.h (for applications that use ISDN symbols)
Category:	system control and tools
Mode:	synchronous
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ❑ Analog

■ Description

The **gc_GetParm( )** function retrieves the parameter value specified by the **parm_id** parameter for a line device. The application can retrieve only one parameter value at a time.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
linedev:	GlobalCall line device handle
parm_id:	The parameter ID definitions are listed in *Table 36. Parameter Descriptions, gc_GetParm( ) and gc_SetParm( )* in the **gc_SetParm( )** function description section. The "Level" column lists whether the parameter is a channel level parameter or a trunk level parameter. To get a trunk level parameter, the **linedev** parameter must be the device ID associated with a network interface trunk.
valuep:	The address of the buffer where the requested information will be stored, see *paragraph 5.5. GC_PARM* for data structure details.

Termination Event: None.

■ Cautions

None

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

int get_calling_party(LINEDEV ldev, char *callerp)
{
   GC_PARM        parm_val;         /* value of parameter returned */
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */

   /*
    * Retrieve calling party number for E-1 CAS line device and print it.
    */
   if (gc_GetParm(ldev, GCPR_CALLINGPARTY, &parm_val) == GC_SUCCESS) {
      strcpy(callerp, parm_val.paddress);
      printf ("Calling party No. is %s\n", parm_val.paddress);
   }
   else {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               metaevent.ldev, gc_error, msg);
      return(gc_error);
   }
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_SetParm( )**

Name: int gc_GetUsrAttr(linedev, usr_attrp)

Inputs: LINEDEV linedev • GlobalCall line device handle
void **usr_attrp • pointer to location where user attribute info will be stored

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: system control and tools

Mode: synchronous

Technology: ■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit
■ Analog

■ Description

The **gc_GetUsrAttr( )** function retrieves the attribute established previously for the line device by the **gc_SetUsrAttr( )** or **gc_OpenEx( )** function.

Parameter	Description
linedev:	GlobalCall line device handle
usr_attrp:	address of the location where the returned attribute information will be stored. This parameter will be set to NULL if the user attribute was not previously set using the **gc_SetUsrAttr( )** or **gc_OpenEx( )** function.

Termination Event: None.

■ Cautions

None

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
```

```
#include <gclib.h>
#include <gcerr.h>

#define MAXCHAN   30                /* max. number of channels in system */
/*
 * Data structure which stores all information for each line
 */
struct linebag {
   LINEDEV  ldev;                   /* GlobalCall line device handle */
   CRN      crn;                    /* GlobalCall API call handle */
   int      state;                  /* state of first layer state machine */
} port[MAXCHAN+1];

/*
 * Retrieves port_num that was set for this device
 * in set_usrattr (gc_SetUsrAttr())
 */

int get_usrattr(LINEDEV ldev, int *port_num)
{
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */
   void           *vattrp;          /* to retrieve the attribute */

   /*
    * Assuming that a line device is opened already and
    * that its ID is ldev, let us retrieve the attribute set
    * for this ldev, previously set by the user using gc_SetUsrAttr()
    */

   if ( gc_GetUsrAttr( ldev, &vattrp) != GC_SUCCESS ) {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               ldev, gc_error, msg);
      return(gc_error);
   }

   *port_num = (int) vattrp;
   /*
    * Processing may continue using this retrieved attribute
    */
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

See Also

- **gc_SetUsrAttr( )**
- **gc_OpenEx( )**

Name:	int gc_GetVer(linedev, releasenump, intnump, component)	
Inputs:	LINEDEV linedev	• GlobalCall line device handle
	unsigned int *releasenump	• pointer to location where production release number will be stored
	unsigned int *intnump	• pointer to location where internal release number will be stored
	long component	• system component
Returns:	0 if successful <0 if failure	
Includes:	gclib.h gcerr.h	
Category:	optional feature	
Mode:	synchronous	
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ■ Analog	

■ Description

The **gc_GetVer( )** function gets version number of specified software component. A Dialogic version number consists of two parts that provide:

- The release type; i.e. production or beta
- The release number, which consists of different elements, depending on the type of release.

  e.g., 1.00 Production
  1.00 Beta 5

The **gc_GetVer( )** function returns the software version number as a long integer (32 bits) in BCD (binary coded decimal) format. *Figure 6* shows the format of the version number that is returned. Each section in the diagram represents a nibble (4 bits).

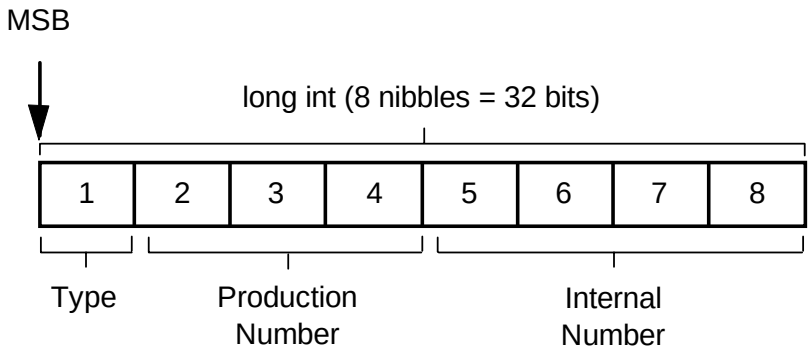

Figure 6. Component Version Number Format

Nibble 1 returns the type of release. The values convert to:

0 - Production
1 - Beta
4 - Special

Nibbles 2, 3, and 4 return the Production Number.

NOTE: Nibbles 2 through 4 are used in all version numbers. Nibbles 5 through 8 only contain values if the release is not a production release.

Nibbles 5, 6, 7, and 8 return the Internal Number, which is used for pre-production product releases. The Internal Number is assigned to beta product releases. Nibbles 5 and 6 hold the product's Beta number. Nibbles 7 and 8 hold additional information that is used for internal releases.

Table 30 provides the values returned by each nibble in the long int. For example, if a production version number is 1.02, then:

*releasenump = 0x0102
*intnump = 0x0000

For a version number of 1.02 beta 2, then:

*releasenump = 0x1102
*intnump = 0x0200

Table 30. gc_GetVer( ) Return Values

*releasenump			*intnump	
1†	**2**†	**3 & 4**†	**5 & 6**†	**7 & 8**†
Type	**Production Number**		**Internal Number**	
Production	Major Prod. Number	Minor Prod. Number	N/A	N/A
Beta	Major Prod. Number	Minor Prod. Number	Beta No.	N/A
† = Nibble(s) [4 bits each]				

Parameter	Description
linedev:	GlobalCall line device handle. If this parameter is set to 0, the version number of the GlobalCall API is returned.
releasenump:	pointer to the location where the production release number and type indicator will be stored.
intnump:	pointer to the location where the internal release number will be stored.
component:	specifies the software component to which the version number applies. Selections are: GCGV_LIB GlobalCall library ICGV_LIB ICAPI library ANGV_LIB ANAPI library ISGV_LIB ISDN library

Termination Event: None

■ Cautions

None

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

int print_version(LINEDEV ldev, long component)
{
   unsigned int   releasenum;       /* Production release number */
   unsigned int   intnum;           /* Internal release number */
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */

   /*
    * Get the version number of the library associate with the line
    * device.
    */
   if (gc_GetVer(ldev, &releasenum, &intnum, component) == GC_SUCCESS) {
      printf("Production release number = 0x%lx\n", releasenum);
      printf("Internal release number = 0x%lx\n", intnum);
   } else {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               ldev, gc_error, msg);
      return(gc_error);
   }
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- None

Name: int gc_GetVoiceH(linedev, voicehp)

Inputs: LINEDEV linedev
- GlobalCall line device handle

int *voicehp
- pointer to returned voice device handle

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: interface specific

Mode: synchronous

Technology: ❑ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit
■ Analog

■ Description

The **gc_GetVoiceH( )** function returns the voice device handle associated with the specified line device, **linedev**. The ***voicehp** parameter is actually the SRL handle of the voice resource associated with the line device. The ***voicehp** parameter can be used as an input to functions requiring a voice handle, such as the voice library's **dx_play( )** function.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
linedev:	GlobalCall line device handle
voicehp	address at which the voice device handle of the voice resource associated with the GlobalCall line device, **linedev,** will be stored

Termination Event: None.

■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

/*
 * Assume the following has been done:
 *    1. A line device has been opened specifying voice resource
 *    2. A call associated with ldev is in the connected state
 */
int get_voice_handle(LINEDEV ldev, int *voicehp)
{
   int           cclibid;          /* cclib id for gc_ErrorValue() */
   int           gc_error;         /* GlobalCall error code */
   long          cc_error;         /* Call Control Library error code */
   char          *msg;             /* points to the error message string */


   if (gc_GetVoiceH(ldev, voicehp) == GC_SUCCESS) {
      /*
       * Application may now perform voice processing (e.g., play a prompt)
       * using the voice handle.
       */
      return(0);
   } else {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               ldev, gc_error, msg);
      return(gc_error);
   }
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

See Also

- **gc_GetNetworkH( )**

Name:	int gc_LoadDxParm(ldev, filepathp, msbufferp, msglength)	
Inputs:	LINEDEV ldev	• line device
	char *filepathp	• pointer to parameter file
	char *msbufferp	• pointer to error message
	int msglength	• maximum error message length
Returns:	0 if successful	
	<0 if failure	
Includes:	gclib.h	
	gcerr.h	
Category:	interface specific	
Mode:	synchronous	
Technology:	❑ ISDN PRI ❑ E-1 CAS ❑ T-1 robbed bit	
	■ Analog	

■ Description

The **gc_LoadDxParm( )** function sets voice parameters associated with a line device that operates as a dedicated or shared resource in conjunction with an analog loop start network interface resource to handle call processing activities. The parameters set by this function affect basic and enhanced call progress and interact with the **gc_MakeCall( )** function.

GlobalCall assigns a LDID number to represent the physical devices that will handle a call, such as a voice resource and an analog loop start (or a digital) network interface resource, when the **gc_Open( )** or **gc_OpenEx( )** function is called. This identification number assignment remains valid until the **gc_Close( )** function is called to close the line devices**.**

When the **gc_LoadDxParm( )** function is called, the function looks for a voice parameter file listing only the voice parameters to be changed from their default value in the location defined by the **filepathp** parameter, typically in the current directory or in the */usr/dialogic/cfg* directory. The voice parameter file is read and the voice device is configured based on the parameters and parameter values defined in this file. Any parameter not defined will use the default parameter value.

The following is an example of a voice channel parameter (*.vcp*) file called *dxchan.vcp* (file names are defined by the user):

```
#
#     D/xxx parameter file for downloading channel level
#     parameters supported by dx_setparm() and DX_CAP structure.
#
#     Values are in decimal unless a leading 0x is included in which
#     case the value is hexadecimal.
#
#     Refer to the Dialogic Voice Software Reference for the DX_CAP
#     structure ca_* parameters. The upper case parameters can also
#     be found in the Dialogic Voice Software Reference under the
#     dx_setparm() function.
#
#     To set a parameter, uncomment (delete the '#' or ';' and set a
#     value to the right of the parameter name.
#

# ca_stdely
# ca_cnosig
# ca_lcdly
# ca_lcdly1
# ca_hedge
# ca_cnosil
# ca_logltch

#
# Values of bitmask flags for setting ca_intflg
# add desired flags to set ca_intflg
#

# DX_OPTEN         =   1
# DX_OPTDIS        =   2
# DX_OPTNOCON      =   3
# DX_PVDENABLE     =   4
# DX_PVDOPTEN      =   5
# DX_PVDOPTNOCON   =   6
# DX_PAMDENABLE    =   7
# DX_PAMDOPTEN     =   8

ca_intflg  5

#
#
#

# ca_lowerfrq
# ca_upperfrq
# ca_timefrq
# ca_maxansr
# ca_ansrdgl
# ca_mxtimefrq
# ca_lower2frq
# ca_upper2frq
# ca_time2frq
# ca_mxtime2frq
# ca_lower3frq
# ca_upper3frq
# ca_time3frq
# ca_mxtime3frq
# ca_dtn_pres
# ca_dtn_npres
# ca_dtn_deboff
# ca_pamd_failtime
# ca_pamd_minring
# ca_pamd_spdval
# ca_pamd_qtemp
```

```
# ca_noanswer
# ca_maxintering

#
# Channel level parameters set by dx_setparm()
#

# DXCH_DFLAGS
# DXCH_DTINITSET
# DXCH_DTMFTLK
# DXCH_DTMFDEB
# DXCH_MFMODE
# DXCH_MAXRWINK
# DXCH_MINRWINK
# DXCH_WINKDLY
# DXCH_RINGCNT
# DXCH_WINKLEN
# DXCH_PLAYDRATE
# DXCH_RECRDRAT
```

A voice parameter file contains parameter definition lines and may contain comment lines. Each parameter definition line comprises:

- a case-sensitive voice parameter as defined in *Table 31. Voice* Channel-level Parameters or *Table 32. Voice Call* Analysis Parameters as the first field of the line, a space and
- a second field defining the parameter value:
  - for voice channel parameter values; see the *Voice Board Parameter Defines for dx_getparm( ) and dx_setparm( ) paragraph* in the *Voice Software Reference, Data Structures and Device Parameters chapter*.
  - for DX_CAP data structure field parameter values; see the *DX_CAP - change default call analysis parameters paragraph* in the *Voice Software Reference, Data Structures and Device Parameters chapter*.

    The parameter value may be entered as a decimal value or as a hexadecimal value when prefixed with a “0x”.

A comment line begins with a:

- # character or a
- ; character.

The **gc_LoadDxParm( )** function will return upon the first detected error. The reason for the error, typically:

- a parsing error (in the *.vcp* file)
- a low-level function call error

- an open file failure error

will be stored in the **msgbufferp** location.

NOTE: Not all errors can be detected by the **gc_LoadDxParm( )** function. Errors in the value of the voice call analysis parameters in the DX_CAP structure cannot be detected until a call is setup by the **gc_MakeCall( )** function.

All channel-level parameters set by the voice function, **dx_setparm( )**, can be set using the **gc_LoadDxParm( )** function. GlobalCall uses the **dx_setparm( )** parameter names to identify all voice channel-level parameters; see *Table 31. Voice* Channel-level Parameters for a summary list of these parameters. Also see the *Voice Board Parameter Defines for dx_getparm( ) and dx_setparm( ) paragraph* in the *Voice Software Reference, Data Structures and Device Parameters chapter* for a description of these parameters.

The **gc_LoadDxParm( )** function supports all basic and enhanced call progress fields defined in the DX_CAP data structure. The call analysis parameters defined in the DX_CAP data structure affect the **gc_MakeCall( )** function. GlobalCall uses the DX_CAP data structure names to identify all call progress parameters; see *Table 32. Voice Call* Analysis Parameters for a summary list of these parameters. Also see the *DX_CAP - change default call analysis parameters paragraph* in the *Voice Software Reference, Data Structures and Device Parameters chapter* for a description of these parameters.

Observe the following criteria when using the **gc_LoadDxParm( )** function:

- Before calling the **gc_LoadDxParm( )** function, use the **gc_Open( ) or gc_OpenEx( )** function to open the voice line device.
- Pass the maximum length of the error message string, **msglength,** to the **gc_LoadDxParm( )** function to avoid overwriting memory locations outside the message string array.

For analog applications, the **gc_LoadDxParm( )** function is used to set board and channel-level parameters previously set by the voice function, **dx_setparm( )**. While the **gc_LoadDxParm( )** function is used for analog applications; the **gc_SetParm( )** and **gc_GetParm( )** functions continue to be used to set and display parameter values for other technologies such as E-1, T-1, ISDN, etc.

Refer also to the *GlobalCall Analog Technology User's Guide* for technology specific information.

Parameter	Description
ldev:	GlobalCall line device handle
filepathp:	specifies a pointer to the voice parameter file to load
msgbufferp:	specifies a pointer to the storage address of any error message
msglength:	defines the maximum length in bytes of the error message stored at address defined by the **msgbufferp** parameter

Table 31. Voice Channel-level Parameters [dx_setparm( )] List

• DXCH_DFLAGS	
• DXCH_DTINITSET	
• DXCH_DTMFDEB	
• DXCH_DTMFTLK	
• DXCH_MAXRWINK	
• DXCH_MFMODE	
• DXCH_MINRWINK	
• DXCH_PLAYDRATE	
• DXCH_RECRDRATE	
• DXCH_RINGCNT	Not used. The default number of rings parameter in the *.cdp* file sets this parameter value.
• DXCH_WINKDLY	
• DXCH_WINKLEN	

Table 32. Voice Call Analysis Parameters (DX_CAP) List

- ca_alowmax
- ca_ansrdgl
- ca_blowmax
- ca_cnosig
- ca_cnosil
- ca_dtn_deboff
- ca_dtn_npres
- ca_dtn_pres
- ca_hedge
- ca_hi1bmax
- ca_hi1ceil
- ca_hi1tola
- ca_hi1tolb
- ca_higltch
- ca_hisiz
- ca_intflg
- ca_lcdly
- ca_lcdly1
- ca_lo1bmax
- ca_lo1ceil
- ca_lo1rmax
- ca_lo2bmax
- ca_lo2rmin

- ca_lo1tola
- ca_lo1tolb
- ca_lo2tola
- ca_lo2tolb
- ca_logltch
- ca_lower2frq
- ca_lower3frq
- ca_lowerfrq
- ca_maxansr
- ca_maxintering
- ca_mxtime2frq
- ca_mxtime3frq
- ca_mxtimefrq
- ca_nbrbeg
- ca_nbrdna
- ca_noanswer
- ca_nsbusy
- ca_pamd_failtime
- ca_pamd_minring
- ca_pamd_qtemp
- ca_pamd_spdval
- ca_stdely
- ca_time2frq
- ca_time3frq
- ca_timefrq

- ca_upper2frq
- ca_upper3frq
- ca_upperfrq

Termination Event: None

■ Cautions

The maximum length of the error message string, **msglength,** should be passed to the function to avoid overwriting memory locations outside the array pointed to by the **msgbufferp** parameter.

The call analysis parameters, see *Table 32. Voice Call* Analysis Parameters, are only used for analog loop start protocols. If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code. See also the *Errors* paragraph at the end of this function description.

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <gclib.h>
#include <gcerr.h>

#define MSGLENGTH 80

main()
{
   LINEDEV ldev;
   char errmsg[MSGLENGTH];
   .
   .
   /*
    *   Assume the following has been done:
    *
    *   Open line device (ldev) specifying voice and network resource using
    *   gc_Open()
    *
    */

   /* call gc_LoadDxParm() to download the channel parameters */
   if ((gc_LoadDxParm(ldev, "dxchan.vcp", errmsg, MSGLENGTH)) != 0) {
      if (gc_error() != EGC_UNSUPPORTED) {
         printf("Error gc_LoadDxParm() loading channel parameters\n");
         printf("%s\n", errmsg);
         exit(1);
```

```
        }
        printf("gc_LoadDxParm() unsupported\n");
        exit(2);
        }
    .
    .
    .
}
```

■ Errors

If this function returns a <0 to indicate failure or if the GCEV_TASKFAIL event is received, use **gc_ErrorValue( )** and the **gc_ResultMsg( )** function as described in *section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

If the error is not EGC_UNSUPPORTED, then a more detailed description of the error is copied to the address specified by the **msgbufferp** parameter. When a parsing error is detected, an "Invalid line" followed by the line number and the line containing the error are stored in the **msgbuffer** buffer.

■ See Also

- **gc_MakeCall( )**
- **gc_Open( )** or **gc_OpenEx( )**

Name:	int gc_MakeCall(linedev, crnp, numberstr, makecallp, timeout, mode)	
Inputs:	LINEDEV linedev	• line device
	CRN *crnp	• pointer to returned call reference number
	char *numberstr	• destination phone number
	GC_MAKECALL_BLK *makecallp	• pointer to outbound call info
	int timeout	• time-out value
	unsigned long mode	• async or sync
Returns:	0 if successful <0 if failure	
Includes:	gclib.h gcerr.h gcisdn.h (for applications that use ISDN symbols)	
Category:	basic call control	
Mode:	asynchronous or synchronous	
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ■ Analog	

■ Description

The **gc_MakeCall( )** function enables the application to make an outgoing call on the specified line device. When this function is issued asynchronously, a CRN will be assigned and returned immediately if the function is successful. All subsequent communications between the application and the GlobalCall library regarding that call will use the CRN as a reference. If this function is issued synchronously, the CRN will be available at the successful completion of the function.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
linedev:	GlobalCall line device handle
crnp:	pointer to the memory location where the CRN is to be stored.
numberstr:	called party's telephone number (must be terminated with '\0'). Maximum length: 32 digits

Parameter	Description
makecallp:	specifies a pointer to the GC_MAKECALL_BLK structure, see *paragraph 5.3. GC_MAKECALL_BLK* for details. Assigning a NULL to this parameter indicates that the default value should be used for the call.
timeout:	time interval (in seconds) during which the call must be established, or function will return with a time-out error. This parameter is ignored when set to 0. Not all call control libraries support this argument in asynchronous mode. Refer to the appropriate *GlobalCall Technology User's Guide* for technology specific information.
mode:	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

If the **gc_MakeCall( )** function fails, the call state may differ depending upon the point in the calling process where the failure occurred and the call control library used.

In the asynchronous mode, if the function is successfully initiated but connection is not achieved (no GCEV_CONNECTED event returned), then the application must issue **gc_DropCall( )** and **gc_ReleaseCall( )** functions to terminate the call completely.

In the synchronous mode, if the ***crnp** is zero, the call state is Null. A Null state indicates that the call was fully terminated and that another **gc_MakeCall( )** function can be issued. For non-zero ***crnp** values, the application or thread (Windows NT only) must issue **gc_DropCall( )** and **gc_ReleaseCall( )** functions to terminate the call completely before issuing another **gc_MakeCall( )** function.

The GC_MAKECALL_BLK structure is a list of parameters used to specify the outbound call.

The GCEV_ALERTING event (enabled by default) notifies the application that the call has reached its destination but is not yet connected to the called party. When this event is received, the call state changes to Alerting. In the Alerting state, the reception of a GCEV_CONNECTED event (or, if in synchronous mode,

the successful completion of the function) causes a transition to the Connected state thus indicating a complete call connection.

The GCEV_CALLSTATUS event informs the application that a timeout or a no answer (call control library dependent) condition occurred. This event does not cause any state change. Not all call control libraries generate this event (e.g., ISDN library).

If glare handling is not specified in the protocol, the inbound call prevails when glare occurs.

The following table lists error conditions, associated event/return values and the result/error value returned. For all errors, the following apply:

- Asynchronous: When an error condition is encountered, an event value such as GCEV_TASKFAIL, GCEV_CALLSTATUS or GCEV_DISCONNECTED is returned. Issue a **gc_ResultValue( )** function to retrieve the reason or result code for the event and then issue a **gc_ResultMsg( )** function to retrieve the ASCII message describing the error condition. When an error condition occurs in asynchronous mode, you must issue the **gc_DropCall( )** and **gc_ReleaseCall( )** functions before you can initiate your next call.
- Synchronous: When an error condition is encountered, a <0 value is returned. Issue a **gc_ErrorValue( )** function to retrieve the error code and then issue a **gc_ResultMsg( )** function to retrieve the ASCII message describing the error condition.

When an error condition occurs in synchronous mode, if the **crn** returned is:

- 0, then the call state is Null; you may initiate your next call or call related operation.
- non-0, then you must issue the **gc_DropCall( )** and **gc_ReleaseCall( )** functions before you can initiate your next call or call related operation.

In asynchronous mode, when the function fails to start, <0 is returned. In this case, no CRN was assigned to the call and you should not do a drop and release call.

Condition	Event/Return Value	Result/Error Value
Call answered at remote end	Async: GCEV_CONNECTED Sync: 0	None - normal completion of function; line is connected and called party answered
Error occurs prior to dialing	Async: GCEV_TASKFAIL Sync: <0	Varies depending on the reason for the failure
Error occurs during dialing	Async: a call control library related error or GCEV_DISCONNECTED Sync: <0	Async: GCRV_TIMEOUT or GCRV_PROTOCOL result value Sync: EGC_TIMEOUT or EGC_PROTOCOL error depending on the call control library used
Busy line	Async: GCEV_DISCONNECTED Sync: <0	Async: GCRV_BUSY result value Sync: EGC_BUSY error
Ring, no answer	Async: GCEV_CALLSTATUS Sync: <0 Not all call control libraries generate this event (e.g., ISDN library).	Async: GCRV_NOANSWER or GCRV_TIMEOUT result value Sync: EGC_NOANSWER or EGC_TIMEOUT error depending on the call control library used
Other errors	Async: reflects the error encountered and the call control library used Sync: <0	Varies depending on the reason for the failure and the call control library used

Termination Event: In the asynchronous mode, if the call results in a successful connection, a GCEV_CONNECTED event is sent to the application; otherwise, a GCEV_TASKFAIL or GCEV_DISCONNECTED event is sent.

■ Cautions

In both asynchronous and synchronous mode, after a timeout or a no answer condition is reported and before the **gc_DropCall( )** function has successfully completed, a GCEV_CONNECTED event may arrive. Ignore this event since the call cannot be salvaged.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>

#define MAXCHAN   30                /* max. number of channels in system */

/*
 * Data structure which stores all information for each line
 */
struct linebag {
   LINEDEV  ldev;                   /* GlobalCall line device handle */
   CRN      crn;                    /* GlobalCall API call handle */
   int      state;                  /* state of first layer state machine */
} port[MAXCHAN+1];
struct linebag *pline;              /* pointer to access line device */

/*
 * Assume the following has been done:
 *    1. Opened line devices for each time slot on DTIB1.
 *    2. Each line device is stored in linebag structure "port"
 */
int make_call(int port_num)
{
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */


   /* Find info for this time slot, specified by 'port_num' */
   /* (assumes port_num is valid) */
   pline = port + port_num;

   /*
    * Make a call to the number 993-3000.
    */
   if ( gc_MakeCall(pline->ldev, &pline->crn, "9933000", NULL, 0, EV_SYNC)
       == GC_SUCCESS) {
      /* Call successfully connected; continue processing */
   }
   else {

      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               pline -> ldev, gc_error, msg);
```

```
      return(gc_error);
   }
   /*
    * Application may now wait for an event to indicate call
    * completion.
    */
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure or if the GCEV_TASKFAIL, GCEV_CALLSTATUS or GCEV_DISCONNECTED event is received, use **gc_ErrorValue( )** or **gc_ResultValue( )**, respectively, and the **gc_ResultMsg( )** function as described in *section 3.11. Error Handling* to retrieve the reason for the error. See the above description for more details on handling errors associated with making a call. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_DropCall( )**
- **gc_LoadDxParm( )**
- **gc_ReleaseCall( )**

Name:	int gc_Open(linedevp, devicename, rfu)	
Inputs:	LINEDEV *linedevp	• pointer to returned line device
	char *devicename	• pointer to ASCII string
	int rfu	• reserved for future use
Returns:	0 if successful	
	<0 if failure	
Includes:	gclib.h	
	gcerr.h	
Category:	system controls and tools	
Mode:	synchronous	
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit	
	■ Analog	

■ Description

The **gc_Open( )** function opens a GlobalCall device and returns a unique line device ID (or handle) to identify the physical device or devices that carry the call (e.g., a line device may represent a single network, time slot or the grouping together of a time slot and a voice channel). All subsequent references to the opened device must be made using the line device ID. After the successful return of the **gc_Open( )** function, the application must wait for a GCEV_UNBLOCKED event before proceeding with a call (make or wait call) on the opened line device. When the GCEV_UNBLOCKED event is received, then the line is ready to accept calls.

NOTE: When you issue a **gc_Open( )** call, you may immediately get a GCEV_UNBLOCKED event before the function returns. This event may be lost unless:

- typically, in a Windows NT environment, event processing 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 to be used, such as when an application is being ported from UNIX, then you must use the asynchronous internal-thread callback model or the asynchronous worker-thread callback model. See *paragraph 3.2. Windows NT Programming Models* for details and the following summaries.

- for Windows NT synchronous mode applications, when using an event handler for GCEV_BLOCKED and GCEV_UNBLOCKED events,

enable the event handler BEFORE creating the threads to handle each channel. (Ensure that the **linedevp** parameter passed to the **gc_Open( )** function is a global variable that can be accessed by your handler.)

- for Windows NT asynchronous mode applications, when the application will handle the events, the automatic creation of an SRL event handling thread can be disabled by setting the **sr_setparm( )** function **parmno** parameter SR_MODELTYPE value to SR_STASYNC so that the event is held in the event queue OR the application can enable an event handler for GCEV_BLOCKED and GCEV_UNBLOCKED events BEFORE opening each channel.

- when using UNIX in signal mode, enable an event handler for any device, any event OR for any device and GCEV_BLOCKED and GCEV_UNBLOCKED events before calling the **gc_Open( )** function OR you can wrap the **gc_Open( )** function with **sr_hold( )** and **sr_release( )** functions (this approach enables setting the user attributes with the **gc_SetUsrAttr( )** function before opening a device).

Both network board and channel (i.e., time slot) devices can be opened using the **gc_Open( )** function**.** A device may only be opened once and cannot be re-opened by the current process or by any other process until the device is closed.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
linedevp:	pointer to unique number to be filled in by this function to identify a specific device
devicename:	pointer to an ASCII string that defines the device(s) associated with the returned **linedevp** number. The **devicename** parameter specifies the device to be opened and the protocol to be used. The format used to define **devicename** is: <field1><field2>...<fieldn> These fields may be listed in any order. The format for a field is: :<key>_<field name> Valid keys and their appropriate field names are:

Parameter	Description

key	field name
P	*protocol_name*
N	*network_device_name*
V	*voice_device_name*

The *protocol_name* field specifies the protocol to be used. Refer to the appropriate *GlobalCall Technology User's Guide* for technology specific protocol information.

The *network_device_name* and *voice_device_name* follow the standard Dialogic naming convention:

- The *network_device_name* field specifies the board name and the time slot name (if needed). If the board is to be opened, the *network_device_name* is the board name in the format:

  dtiB<number of board>

  If a time slot is to be opened, both the board and time slot are specified in the format:

  dtiB<number of board>T<time slot number>

- The *voice_device_name* specifies the voice board and channel:

  dxxxB<virtual board number>C<channel number>

The *voice_device_name* field is not valid when opening an ISDN device.

See the *GlobalCall Technology User's Guide* for your technology for information about which fields to use when opening a device.

rfu: reserved for future use. Set **rfu** to 0.

Termination Event: None.

■ Cautions

If a handler is enabled for the GCEV_UNBLOCKED event, then the **linedevp** parameter passed to the **gc_Open( )** function must be global.

When you issue a **gc_Open( )** call, you immediately get a GCEV_UNBLOCKED event. This event may be lost unless your application is structured to capture this event when you open each channel, see Note above for details.

To handle error returns from the **gc_Open( )** function, use the GlobalCall error handling functions, **gc_ErrorValue( )** and **gc_ResultMsg( )**. Do not use the UNIX **errno** variable to get GlobalCall error information.

See the *GlobalCall Technology User's Guide* for your network interface to determine required **devicename** components and features unique to your network interface such as voice resource usage.

■ Example

UNIX example: the following example code illustrates opening multiple line devices using the UNIX signal mode.

```
/*
 * Standard Dialogic header(s)
 */
#include <srllib.h>
#include <dxxxlib.h>
#include <dtilib.h>

/*
 * GlobalCall header(s)
 */
#include <gclib.h>
#include <gcerr.h>

#define MAXCHAN   30              /* max. number of channels in system */

/*
 * Global variable
 */
METAEVENT metaevent;
char *program_name;               /* program name */

/*
 * Function prototype(s)
 */
int print_error(char *function);
```

```
int evt_hdlr(void);
int open_line_devices(void);
int close_line_devices(void);

/*
 * Data structure which stores all information for each line
 */
static struct channel {
   LINEDEV  ldev;                  /* GlobalCall API line device handle */
   CRN      crn;                   /* GlobalCall API call handle */
   int      blocked;               /* channel blocked/unblocked */
   int      networkh;              /* network handle */
   int      voiceh;                /* voice handle */
} port[MAXCHAN+1];

/*
 * Main Program
 */
void main(int argc, char *argv[])
{
  int mode;

  /* Compiler warning */
  program_name = argv[0];
  argc = argc;

  /* Set SRL mode */
  mode = SR_POLLMODE;
  if (sr_setparm(SRL_DEVICE, SR_MODEID, &mode) == -1) {
     printf( "Unable to set to Polled Mode");
     exit(1);
  }

  /* Enable the event handler */
  if (sr_enbhdlr(EV_ANYDEV, EV_ANYEVT,
      (long (*) (void *))evt_hdlr) == -1) {
      printf("sr_enbhdlr failed\n");
      exit(1);
   }

   /* Start the library */
   if (gc_Start(NULL) != GC_SUCCESS) {
      /* process error return as shown */
      print_error("gc_Start");
   }

   /* open the line devices */
   open_line_devices();

   sr_waitevt(50);

   /* close the line devices */
   close_line_devices();

   /* Stop the library */
   if (gc_Stop() != GC_SUCCESS) {
      /* process error return as shown */
      print_error("gc_Stop");
   }
}

/*
 * int print_error (char *function)
 *
 * INPUT: char *function - function name
```

```
 * RETURN: gc_error      - globalcall error number
 *
 */
int print_error(char *function)
{
   int            cclibid;       /* cclib id for gc_ErrorValue() */
   int            gc_error;      /* GlobalCall error code */
   long           cclib_error;   /* Call Control Library error code */
   char           *gcmsg;        /* points to the gc error message string */
   char           *ccmsg;        /* points to the cclib error message string */

   gc_ErrorValue( &gc_error, &cclibid, &cclib_error);
   gc_ResultMsg( LIBID_GC, (long) gc_error, &gcmsg);
   gc_ResultMsg( cclibid, cclib_error , &ccmsg);
   printf ("gc_Open failed, gc(0x%lx) - %s, cc(0x%lx) - %s\n",
            gc_error, gcmsg, cclib_error, ccmsg);

   return (gc_error);
}

/*
 * int evt_hdlr (void)
 *
 * RETURN: 0             - function successful
 *         error         - GlobalCall error number
 *
 */
int evt_hdlr(void)
{
   struct channel *pline;
   int error;                       /* reason for failure of function */

   if (gc_GetMetaEvent(&metaevent) != GC_SUCCESS) {
      /* process error return as shown */
      error = print_error("gc_GetMetaEvent");
      return(error);
   }

   if (metaevent.flags & GCME_GC_EVENT) {
      /* process GlobalCall events */

      if (gc_GetUsrAttr(metaevent.linedev, (void **)&pline) != GC_SUCCESS) {
         /* process error return as shown */
         error = print_error("gc_GetUsrAttr");
         return(error);
      }

      switch (metaevent.evttype) {
         case GCEV_UNBLOCKED:
            printf("received GCEV_UNBLOCKED event on %s\n",
                   ATDV_NAMEP(pline->networkh));
            pline->blocked = 0;
         break;

         default:
            printf ("Unexpected GlobalCall event received\n");
         break;
      }
   }
   else {

       /* process other events */
   }

   return 0;
```

```
}

/*
 * int open_line_devices (void)
 *
 * RETURN: 0             - function successful
 *         error         - GlobalCall error number
 *
 */
int open_line_devices(void)
{
   char           devname[64];         /* argument to gc_Open() function */
   int            vbnum = 0;           /* virtual board number (1-based) */
   int            vch = 0;             /* voice channel number (1-based) */
   int            ts;                  /* time slot number (1-based) */
   int            port_index;          /* index for 'port' */
   int            lines, brds, tslots; /* variables used for voice/net lib calls */
   int            error;               /* reason for failure of function */

   /*
    * Construct device name parameter for Open function and
    * Opened line devices for each time slot on DTIB1 using inbound
    * Argentina R2 protocol.
    */
   for (ts = 1,port_index = 1; ts <= MAXCHAN; ts++,port_index++) {

      vbnum = (ts - 1) / 4 + 1;
      vch = ((ts - 1) % 4) + 1;
      sprintf (devname, ":N_dtiB1T%d:P_ar_r2_o:V_dxxxB%dC%d", ts, vbnum, vch);
      sr_hold();
      if (gc_Open(&port[port_index].ldev, devname, 0) != GC_SUCCESS) {
         /* process error return as shown */
         error = print_error("gc_Open");
         sr_release();
         return(error);
      }

      if (gc_SetUsrAttr(port[port_index].ldev,
          (void *)&port[port_index]) != GC_SUCCESS) {
         /* process error return as shown */
         error = print_error("gc_SetUsrAttr");
         sr_release();
         return(error);
      }

      if (gc_GetNetworkH(port[port_index].ldev,
          &(port[port_index].networkh)) != GC_SUCCESS) {
         /* process error return as shown */
         error = print_error("gc_GetNetworkH");
         sr_release();
         return(error);
      }

      if (gc_GetVoiceH(port[port_index].ldev,
          &(port[port_index].voiceh)) != GC_SUCCESS) {
         /* process error return as shown */
         error = print_error("gc_GetVoiceH");
         sr_release();
         return(error);
      }

      port[port_index].blocked = 1;  /* channel is blocked until unblocked */
                                     /* event is received. */

      sr_release();
```

```
   }

   /*
    * Application is now ready to make a call or wait for a call.
    */
   return (0);
}

/*
 * int close_line_devices (void)
 *
 * RETURN: 0             - function successful
 *         error         - GlobalCall error number
 *
 */
int close_line_devices(void)
{
   int port_index;    /* port index */
   int error;         /* reason for failure of function */

   for (port_index = 1; port_index <= MAXCHAN; port_index++) {
      if (gc_Close(port[port_index].ldev) != GC_SUCCESS) {
         /* process error return as shown */
         error = print_error("gc_Close");

         return (error);
      }
   }

   if (sr_dishdlr(EV_ANYDEV, EV_ANYEVT,
       (long (*) (void *))evt_hdlr) == -1) {
       printf("sr_dishdlr failed\n");
       exit(1);
   }

   return;
}
```

Windows NT example: the following example illustrates enabling an event handler before issuing the **gc_Open( )** function to capture the GCEV_UNBLOCKED event when using Windows NT multithreaded applications.

```
/*
 * Windows header(s)
 */
#include <windows.h>

/*
 * Standard Dialogic header(s)
 */
#include <srllib.h>
#include <dxxxlib.h>
#include <dtilib.h>

/*
 * GlobalCall header(s)
 */
#include <gclib.h>
#include <gcerr.h>
```

```
#define MAXCHAN   30               /* max. number of channels in system */

/*
 * Data structure which stores all information for each line
 */
static struct channel {
   LINEDEV  ldev;                  /* GlobalCall API line device handle */
   CRN      crn;                   /* GlobalCall API call handle */
   int      blocked;               /* channel blocked/unblocked */
   int      networkh;              /* network handle */
   int      voiceh;                /* voice handle */
} port[MAXCHAN+1];

/*
 * Global variable(s)
 */
METAEVENT metaevent;              /* metaevent structure */
char *program_name;               /* program name */

/*
 * Function prototype(s)
 */
int print_error(char *function);
int evt_hdlr(void);
int open_line_devices(void);
int close_line_devices(void);

/*
 * Main Program
 */
void main(int argc, char *argv[])
{
   /* Set srl mode */
   int mode = SR_STASYNC;

   /* Compiler warnings */
   program_name = argv[0];
   argc = argc;

   /* Set SRL mode */
   sr_setparm(SRL_DEVICE, SR_MODELTYPE, &mode);

   /* Enable the event handler */
   if (sr_enbhdlr(EV_ANYDEV, EV_ANYEVT,
      (long (*) (unsigned long ))evt_hdlr) == -1) {
      printf("sr_enbhdlr failed\n");
      exit(1);
   }

   /* Start the library */
   if (gc_Start(NULL) != GC_SUCCESS) {
      /* process error return as shown */
      print_error("gc_Start");
   }

   /* open the line devices */
   open_line_devices();

   /* wait for an event */
   sr_waitevt(50);

   /* close the line devices */
   close_line_devices();

   /* Stop the library */
```

```
   if (gc_Stop() != GC_SUCCESS) {
      /* process error return as shown */
      print_error("gc_Stop");
   }
}

/*
 * int print_error (char *function)
 *
 * INPUT: char *function - function name
 * RETURN: gc_error      - GlobalCall error number
 *
 */
int print_error(char *function)
{
   int            cclibid;       /* cclib id for gc_ErrorValue() */
   int            gc_error;      /* GlobalCall error code */
   long           cclib_error;   /* Call Control Library error code */
   char           *gcmsg;        /* points to the gc error message string */
   char           *ccmsg;        /* points to the cclib error message string */

   gc_ErrorValue( &gc_error, &cclibid, &cclib_error);
   gc_ResultMsg( LIBID_GC, (long) gc_error, &gcmsg);
   gc_ResultMsg( cclibid, cclib_error , &ccmsg);
   printf ("%s failed\n gc(0x%lx) - %s\n cc(0x%lx) - %s\n",
            function, gc_error, gcmsg, cclib_error, ccmsg);

   return (gc_error);
}

/*
 * int evt_hdlr (void)
 *
 * RETURN: 0             - function successful
 *         error         - globalcall error number
 *
 */
int evt_hdlr(void)
{
   struct channel *pline;
   int error;                       /* reason for failure of function */

   if (gc_GetMetaEvent(&metaevent) != GC_SUCCESS) {
      /* process error return as shown */
      error = print_error("gc_GetMetaEvent");
      return(error);
   }

   if (metaevent.flags & GCME_GC_EVENT) {
      /* process GlobalCall events */

      if (gc_GetUsrAttr(metaevent.linedev, (void **)&pline) != GC_SUCCESS) {
         /* process error return as shown */
         error = print_error("gc_GetUsrAttr");
         return(error);
      }

      switch (metaevent.evttype) {
         case GCEV_UNBLOCKED:
            printf("received GCEV_UNBLOCKED event on %s\n",
                   ATDV_NAMEP(pline->networkh));
            pline->blocked = 0;
         break;

         default:
```

```
            printf ("Unexpected GlobalCall event received\n");
         break;
      }
   }
   else {

       /* process other events */
   }

   return 0;
}

/*
 * int open_line_devices (void)
 *
 * RETURN: 0             - function successful
 *         error         - GlobalCall error number
 *
 */
int open_line_devices(void)
{
   char           devname[64];   /* argument to gc_Open() function */
   int            vbnum = 0;     /* virtual board number (1-based) */
   int            vch = 0;       /* voice channel number (1-based) */
   int            ts;            /* time slot number (1-based) */
   int            port_index;    /* index for 'port' */
   int            error;         /* reason for failure of function */

   /*
    * Construct device name parameter for Open function and
    * Opened line devices for each time slot on DTIB1 using inbound
    * Brazil R2 protocol.
    */
   for (ts = 1,port_index = 1; ts <= MAXCHAN; ts++,port_index++) {

      vbnum = (ts - 1) / 4 + 1;
      vch = ((ts - 1) % 4) + 1;
      sprintf (devname, ":N_dtiB1T%d:P_br_r2_o:V_dxxxB%dC%d", ts, vbnum, vch);

     /* open line device */
      if (gc_Open(&port[port_index].ldev, devname, 0) != GC_SUCCESS) {
         /* process error return as shown */
         error = print_error("gc_Open");

         return(error);
      }

     /* assign port[port_index].ldev to *dev_handle */
     printf("%ld\n", port[port_index].ldev);

     /* set user attribute */
      if (gc_SetUsrAttr(port[port_index].ldev,
          (void *)&port[port_index]) != GC_SUCCESS) {
         /* process error return as shown */
         error = print_error("gc_SetUsrAttr");

         return(error);
      }

      /* get network handle */
      if (gc_GetNetworkH(port[port_index].ldev,
          &(port[port_index].networkh)) != GC_SUCCESS) {
         /* process error return as shown */
         error = print_error("gc_GetNetworkH");
```

```
        return(error);
      }

     /* get voice handle */
      if (gc_GetVoiceH(port[port_index].ldev,
          &(port[port_index].voiceh)) != GC_SUCCESS) {
         /* process error return as shown */
         error = print_error("gc_GetVoiceH");

         return(error);
      }

      port[port_index].blocked = 1;  /* channel is blocked until unblocked */
                                     /* event is received. */
   }

   /*
    * Application is now ready to make a call or wait for a call.
    */
   return (0);
}

/*
 * int close_line_devices (void)
 *
 * RETURN: 0             - function successful
 *         error         - GlobalCall error number
 *
 */
int close_line_devices(void)
{
   int port_index;    /* port index */
   int error;         /* reason for failure of function */

   for (port_index = 1; port_index <= MAXCHAN; port_index++) {
      /* close line device */
       if (gc_Close(port[port_index].ldev) != GC_SUCCESS) {
          /* process error return as shown */
          error = print_error("gc_Close");

          return (error);
       }
   }

   /* disable the handler */
   if (sr_dishdlr(EV_ANYDEV, EV_ANYEVT,
       (long (*)(unsigned long ))evt_hdlr) == -1) {
       printf("sr_dishdlr failed\n");
       exit(1);
   }

   return 0;
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

See Also

- **gc_Attach( )**
- **gc_Close( )**
- **gc_Detach( )**
- **gc_GetNetworkH( )**
- **gc_GetVoiceH( )**
- **gc_LoadDxParm( )**
- **gc_OpenEx( )**

Name:	int gc_OpenEx(linedevp, devicename, rfu, usrattr)	
Inputs:	LINEDEV *linedevp	• pointer to returned line device
	char *devicename	• pointer to ASCII string
	int rfu	• reserved for future use
	void *usrattr	• pointer to user attribute
Returns:	0 if successful	
	<0 if failure	
Includes:	gclib.h	
	gcerr.h	
Category:	system controls and tools	
Mode:	synchronous	
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit	
	■ Analog	

■ Description

The **gc_OpenEx( )** function opens a GlobalCall device and sets user defined attribute and returns a unique line device ID (or handle) to identify the physical device or devices that carry the call (e.g., a line device may represent a single network, time slot or the grouping together of a time slot and a voice channel).

The **gc_OpenEx( )** function can be used in place of a **gc_Open( )** function followed by a **gc_SetUsrAttr( )** function. The **gc_OpenEx( )** function includes all the functionality of the **gc_Open( )** function (see the **gc_Open( )** function description for details) plus the added feature of the **usrattr** parameter. The **usrattr** parameter points to a buffer where a user defined attribute is stored thus eliminating the need to call the **gc_SetUsrAttr( )** function after calling a **gc_Open( )** function.

Examples of using **usrattr** include using it as a pointer to a data structure associated with a line device or an index to an array. The data structure may contain user information such as the current call state, line device identification, etc.

Parameter	Description
linedevp:	see the **gc_Open( )** function description for details
devicename:	see the **gc_Open( )** function description for details

Parameter	Description
rfu:	see the **gc_Open( )** function description for details
usrattr:	pointer to buffer where a user defined attribute is stored.

Termination Event: None.

■ Cautions

See the **gc_Open( )** function description for details.

■ Example

This **gc_OpenEx( )** function example uses the same example code as the **gc_Open( )** function example except that the open line devices subroutine is replaced with the following subroutine:

```
.
.
.
 */
int open_line_devices(void)
{
   char           devname[64];         /* argument to gc_OpenEx() function */
   int            vbnum = 0;           /* virtual board number (1-based) */
   int            vch = 0;             /* voice channel number (1-based) */
   int            ts;                  /* time slot number (1-based) */
   int            port_index;          /* index for 'port' */
   int            lines, brds, tslots; /* variables used for voice/net lib calls */
   int            error;               /* reason for failure of function */

   /*
    * Construct device name parameter for Open function and
    * Opened line devices for each time slot on DTIB1 using inbound
    * Argentina R2 protocol.
    */
   for (ts = 1,port_index = 1; ts <= MAXCHAN; ts++,port_index++) {

      vbnum = (ts - 1) / 4 + 1;
      vch = ((ts - 1) % 4) + 1;
      sprintf (devname, ":N_dtiB1T%d:P_ar_r2_o:V_dxxxB%dC%d", ts, vbnum, vch);
      sr_hold();
      if (gc_OpenEx(&port[port_index].ldev, devname, 0, (void *)&port[port_index])
         != GC_SUCCESS) {
         /* process error return as shown */
         error = print_error("gc_Open");
         sr_release();
         return(error);
      }
```

```
      /* NOTE: The gc_SetUsrAttr() function is not required because
       *       the user attribute was set as a parameter in the
       *       gc_OpenEx() function.
       */

      if (gc_GetNetworkH(port[port_index].ldev,
          &(port[port_index].networkh)) != GC_SUCCESS) {
         /* process error return as shown */
         error = print_error("gc_GetNetworkH");
         sr_release();
         return(error);
      }

      if (gc_GetVoiceH(port[port_index].ldev,
          &(port[port_index].voiceh)) != GC_SUCCESS) {
         /* process error return as shown */
         error = print_error("gc_GetVoiceH");
         sr_release();
         return(error);
      }

      port[port_index].blocked = 1;  /* channel is blocked until unblocked */
                                     /* event is received. */

      sr_release();
   }

   /*
    * Application is now ready to make a call or wait for a call.
    */
   return (0);
}
.
.
.
```

■ Errors

See the **gc_Open( )** function description for details.

■ See Also

- See the **gc_Open( )** function description for details.
- **gc_GetUsrAttr( )**
- **gc_SetUsrAttr( )**

Name: int gc_ReleaseCall(crn)
Inputs: CRN crn • call reference number
Returns: 0 if successful
<0 if failure
Includes: gclib.h
gcerr.h
Category: basic call control
Mode: synchronous
Technology: ■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit
■ Analog

■ Description

The **gc_ReleaseCall( )** function releases all internal resources for the specified call**.** This function must be called after a **gc_DropCall( )** function completes.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
crn:	Call Reference Number

Termination Event: None.

■ Cautions

Applications should call the **gc_ReleaseCall( )** function to release the CRN after a connection is terminated. Failure to do so may cause memory problems due to memory being allocated and not being released.

After issuing this function, the CRN is no longer valid.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
```

```
#include <gclib.h>
#include <gcerr.h>

/*
 * Assume the following has been done:
 *    1. Opened line devices for each time slot on DTIB1.
 *    2. Wait for a call using gc_WaitCall()
 *    3. An event has arrived and has been converted to a metaevent
 *       using gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows NT)
 *    4. The call has been dropped with gc_DropCall()
 */
int release_call(CRN crn)
{
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */

   /*
    * Release the system resources using gc_ReleaseCall().
    */
   if (gc_ReleaseCall(crn) != GC_SUCCESS) {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               metaevent.evtdev, gc_error, msg);
      return(gc_error);
   }
   /*
    * Once gc_ReleaseCall() returns, system is now ready to generate
    * or accept another call on this line device.
    */
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_AnswerCall( )**
- **gc_DropCall( )**
- **gc_MakeCall( )**
- **gc_WaitCall( )**

Name:	int gc_ReqANI(crn, ani_buf, reqtype, mode)	
Inputs:	CRN crn	• call reference number
	char *ani_buf	• buffer to store ANI digits
	int reqtype	• request type
	unsigned long mode	• async or sync
Returns:	0 if successful	
	<0 if failure	
Includes:	gclib.h	
	gcerr.h	
	gcisdn.h	
Category:	interface specific	
Mode:	asynchronous or synchronous	
Technology:	■ ISDN PRI ❑ E-1 CAS ❑ T-1 robbed bit	
	❑ Analog	

■ Description

The **gc_ReqANI( )** function returns the caller's ID, which is normally included in the ISDN setup message. If the caller ID does not exist, and the (AT&T) *ANI-on-Demand* feature is available, the driver will automatically request caller ID from the network. The returned caller ID is stored in the buffer indicated by the **ani_buf** parameter.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
crn:	Call Reference Number
ani_buf:	address of the buffer where ANI digits will be loaded. This buffer will be terminated by '\0'.
reqtype:	request type; see *Table 33* for possible values
mode:	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

Table 33. ANI Request Types

Request Type	Description
ISDN_CPN_PREF	Calling party number preferred.
ISDN_BN_PREF	Billing number preferred.
ISDN_CPN	Calling party number only.
ISDN_BN	Billing number only.
ISDN_CA_TSC	Special use.

Termination Event: In the asynchronous mode, if the calling party number is acquired successfully, a GCEV_REQANI event is sent to the application; otherwise, a GCEV_TASKFAIL event is sent.

A GCEV_DISCONNECTED event may be an unsolicited event reported to the application after a **gc_ReqANI( )** function is issued.

■ Cautions

Ensure that **ani_buf** buffer is at least as large as GC_ADDRSIZE bytes. Currently, *ANI-on-Demand* is only available on the AT&T ISDN network. If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

■ Example

```
/*
 * Assume the following has been done:
 * 1. device has been opened (e.g. :N_dtiB1T1:P_isdn,
 *    :N_dtiB1T2:P_isdn, etc...)
 * 2. gc_WaitCall() has been issued to wait for a call.
 * 3. gc_MetaEvent() has been called to convert the event into metaevent.
 * 4. a GCEV_OFFERED has been detected.
 */


#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
```

```
#include <gcisdn.h>

/*
 * For this example, let's assume that the mode = EV_SYNC and
 * req_type = ISDN_CPN_PREF (Calling Party Number Preferred).
 * req_type can be one of following:
 *     ISDN_BN_PREF (Billing Number preferred)
 *     ISDN_CPN     (Calling Party Number only)
 *     ISDN_BN      (Billing Number only)
 *     ISDN_CA_TSC  (Special Use)
 */
int req_cpn(CRN crn, char *ani_buf, int req_type, unsigned long mode)
{
  LINEDEV  ddd;             /* Line device */
  int      gc_err;          /* GlobalCall Error Code */
  int      cclibid;         /* Call Control library ID */
  long     cclib_err;       /* Call Control Error Code */
  char     *msg;            /* Error Message */

  if(gc_CRN2LineDev(crn, &ddd) != GC_SUCCESS) {
    gc_ErrorValue(&gc_err, &cclibid, &cclib_err);
    gc_ResultMsg(cclibid, cclib_err, &msg);
    printf ("Error: gc_CRN2LineDev ErrorValue: %d - %s\n",
             cclib_err, msg);
    return(cclib_err);
  }

  if(gc_ReqANI(crn, ani_buf, req_type, mode) != GC_SUCCESS) {
    gc_ErrorValue(&gc_err, &cclibid, &cclib_err);
    gc_ResultMsg(cclibid, cclib_err, &msg);
    printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
             ddd, gc_err, msg);
    return(cclib_err);
  }

  return(0);
}
```

■ Errors

If this function returns a <0 to indicate failure or if the GCEV_TASKFAIL event is received, use **gc_ErrorValue( )** or **gc_ResultValue( )**, respectively, and the **gc_ResultMsg( )** function as described in *section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_GetANI( )**
- **gc_WaitCall( )**

Name:	int gc_ResetLineDev(linedev, mode)
Inputs:	LINEDEV linedev • GlobalCall line device handle unsigned long mode • async or sync
Returns:	0 if successful <0 if failure
Includes:	gclib.h gcerr.h
Category:	system control and tools
Mode:	asynchronous or synchronous
Technology:	■ ISDN PRI ❑ E-1 CAS ❑ T-1 robbed bit ■ Analog

■ Description

The **gc_ResetLineDev( )** function disconnects any active calls on the line device. All calls being setup are aborted. This function is typically used after a recovery from a trunk error, a recovery from an alarm condition or when resetting the channel to the Null state.

When used in asynchronous mode, the GCEV_RESETLINEDEV event indicates successful termination of the **gc_ResetLineDev( )** function. After receiving this event, the application must issue a new **gc_WaitCall( )** function to receive the next incoming call on the channel.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
linedev:	GlobalCall line device
mode:	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

Termination Event: In the asynchronous mode, GCEV_RESETLINEDEV event is sent to application if successful; GCEV_TASKFAIL event if not successful.

■ Cautions

After successful completion of this function, the application must issue a new **gc_WaitCall( )** function to receive the next call on the channel.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#define MAXCHAN   30      /* max. number of channels in system */
/*
 * Data structure which stores all information for each line
 */
struct linebag {
   LINEDEV  ldev;         /* GlobalCall line device handle */
   CRN      crn;          /* GlobalCall API call handle */
   int      state;        /* state of first layer state machine */
} port[MAXCHAN+1];

/*
 * Assume the following has been done:
 *    1. Opened line devices for each time slot on DTIB1.
 *    2. Application has received GCEV_BLOCKED due to an alarm
 *       condition on the line
 *    3. Application has received GCEV_UNBLOCKED due to alarm
 *       recovered
 *
 * At this point, the application can 'reset'
 * all of it's line devices back to normal.
 * (Alternatively, this could be called at any time)
 */

int restart(void)
{
   int      i;            /* index for 'port' */
   int      ts;           /* network time slot number */

   /*
    * Clean up and get ready to generate/accept calls again.
    */
   for (ts = 1,i=1; ts <= MAXCHAN; ts++,i++) {
      if (gc_ResetLineDev(port[i].ldev, EV_SYNC) != GC_SUCCESS) {
         /* get cause value and process error */
      }

      /*
       * Application will need to re-issue gc_WaitCall() to wait
       * for incoming calls
       */
   }
   return (0);
}
```

Errors

If this function returns a <0 to indicate failure or if the GCEV_TASKFAIL event is received, use **gc_ErrorValue( )** or **gc_ResultValue( )**, respectively, and the **gc_ResultMsg( )** function as described in *section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

See Also

- **gc_WaitCall( )**

Name: int gc_ResultMsg(cclibid, result_code, msg)

Inputs:

int cclibid	• call control library ID
long result_code	• used to get associated message
char **msg	• pointer to address of returned message string

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: system control and tools

Mode: synchronous

Technology: ■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ■ Analog

■ Description

The **gc_ResultMsg( )** function retrieves an ASCII string describing result code. The **result_code** parameter may represent an error code returned by the **gc_ErrorValue( )** function or a result value returned by a **gc_ResultValue( )** function.

Parameter	Description
cclibid:	call control library identification from which the **result_code** was generated. If the **result_code** value is a GlobalCall error code or result value, then set **cclibid** to LIBID_GC.
result_code:	result value of the event or error code from the library, **cclibid**
msg:	pointer to address where the description of the **result_code** message will be stored

Termination Event: None.

■ Cautions

Do not overwrite the ***msg** pointer as it points to private internal GlobalCall data space.

Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

/* cclidid = LIBID_GC will print GC Lib's error code */
void print_result_msg(int cclibid, long result_code)
{
   char          *msg;             /* points to the error message string */
   char          *lib_name;        /* library name for cclibid */

   if (gc_ResultMsg(cclibid, result_code, &msg) == GC_SUCCESS) {
      gc_CCLibIDToName(cclibid, &lib_name);
      printf("%A library had error 0x%lx - %s\n", lib_name, cc_error, msg);
   } else {
      printf("gc_ResultMsg failed\n");
   }
}
```

Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

See Also

- **gc_ResultValue( )**

Name: int gc_ResultValue(metaeventp, gc_resultp, cclibidp, cclib_resultp)

Inputs:

METAEVENT *metaeventp	• pointer to a metaevent block
int *gc_resultp	• pointer to returned GlobalCall result
int *cclibidp	• pointer to returned call control library ID
long *cclib_resultp	• pointer to returned call control library result

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h
icapi.h (optional, if using ICAPI errors)
gcisdn.h (optional, if using ISDN errors)

Category: system control and tools

Mode: synchronous

Technology: ■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ■ Analog

■ Description

The **gc_ResultValue( )** function retrieves the cause of an event. The "result" uniquely identifies the cause of the event to which the **metaeventp** parameter points. The GlobalCall result value, the call control library ID, and the actual call control library result value are available upon successful return of the function.

Parameter	Description
metaeventp:	pointer to the event data block. This pointer is acquired via **gc_GetMetaEvent( )** or the **gc_GetMetaEventEx( )** function (Windows NT extended asynchronous mode only)..
gc_resultp:	address where the GlobalCall result value is to be stored
cclibidp:	address where the identification of the call control library associated with this metaevent is to be stored

Parameter	Description
cclib_resultp:	address where the result value associated with the call control library metaevent is to be stored

Termination Event: None.

■ Cautions

None

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

int get_result_value(void)
{
   int           gc_result;       /* GlobalCall error code */
   int           cclibid;         /* Call Control Library ID */
   long          cc_result;       /* Call Control Library error code */
   char          *msg;            /* pointer to error message string */

   /* Obtain  the event data */
   sr_waitevt(-1);                /* Wait indefinitely for an event */
   gc_GetMetaEvent(&metaevent);   /* Get event parameters into metaevent */

   /* find the reason for the event */
   if (gc_ResultValue( &metaevent, &gc_result, &cclibid, &cc_result)
       != GC_SUCCESS) {
      gc_ResultMsg( LIBID_GC, (long) gc_result, &msg);
      printf ("Event 0x%lx received on LDEV: %ld - %s\n", metaevent.evttype,
               metaevent.evtdev, msg);
      return(0);
   } else {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf("Error retrieving ResultValue on line device handle: 0x%lx,\
              ErrorValue: 0x%lx - %s\n",
         metaevent.evtdev, gc_error, msg);
      return(gc_error);
   }
}
```

Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

See Also

- **gc_ResultMsg( )**

Name: int gc_SetBilling(crn, rate_type, ratep, mode)

Inputs:
- CRN crn — • call reference number
- int rate_type — • type of billing data
- GC_RATE_U *ratep — • pointer to call charge rate
- unsigned long mode — • async or sync

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h
gcisdn.h (for applications that use ISDN symbols)

Category: optional feature

Mode: asynchronous or synchronous

Technology: ■ ISDN PRI ■ E-1 CAS ❑ T-1 robbed bit ❑ Analog

■ Description

The **gc_SetBilling( )** function sets billing information for the call associated with the specified CRN. For protocols that support this feature, this function tells the Central Office whether or not to charge for the call. For AT&T ISDN applications, the billing rate is available to applications that use AT&T's Vari-Bill service. For some E-1 CAS protocols, different billing rates can be chosen on a per call basis.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information about the **rate_type, ratep** and **mode** parameters.

Parameter	Description
crn:	Call Reference Number
rate_type:	type of billing data.
ratep:	pointer to a data structure which contains the charge information for the current call.
mode:	set to EV_SYNC for synchronous execution or to EV_ASYNC for asynchronous execution (refer to the appropriate *GlobalCall Technology User's Guide* to determine if the asynchronous mode is supported for your technology)

Termination Event: In the asynchronous mode, a GCEV_SETBILLING event is sent to the application if successful; a GCEV_TASKFAIL event if not successful..

■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

■ Example

```
/*
 * Assume the following has been done:
 * 1. device has been opened (e.g. :N_dtiB1T1:P_isdn,
 *    :N_dtiB1T2:P_isdn, etc...)
 * 2. gc_WaitCall() has been issued to wait for a call.
 * 3. gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows NT) has been
 *    called to convert the event into metaevent.
 * 4. a GCEV_OFFERED has been detected.
 * 5. a call has been established.
 */


#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>

/*
 * For this example, let's assume that mode = SYNC and
 * the rate_type = ISDN_FLAT_RATE. The ratep stores the billing information.
 * rate_type can be one of the following:
 *          ISDN_NEW_RATE
 *          ISDN_PREM_CHARGE
 *          ISDN_PREM_CREDIT
 *          ISDN_FREE_CALL
 *
 * Note: This is only available for some protocols.
 *       This function call is used anytime after the connection
 *       is established.
 */
int set_billing(CRN crn, int rate_type, GC_RATE_U *ratep, unsigned long mode)
{
  LINEDEV  ddd;             /* Line device */
  int      gc_err;          /* GlobalCall Error Code */
  int      cclibid;         /* Call Control library ID */
  long     cclib_err;       /* Call Control Error Code */
  char     *msg;            /* Error Message */

  if(gc_CRN2LineDev(crn, &ddd) != GC_SUCCESS) {
    gc_ErrorValue(&gc_err, &cclibid, &cclib_err);
    gc_ResultMsg(cclibid, cclib_err, &msg);
    printf ("Error: gc_CRN2LineDev ErrorValue: %d - %s\n",
             cclib_err, msg);
```

```
    return(cclib_err);
  }

  if(gc_SetBilling(crn, rate_type, ratep, mode) != GC_SUCCESS) {
    gc_ErrorValue(&gc_err, &cclibid, &cclib_err);
    gc_ResultMsg(cclibid, cclib_err, &msg);
    printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
             ddd, gc_err, msg);
    return(cclib_err);
  }

  return(0);

}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_SetParm( )**

Name: int gc_SetCallingNum(linedev, calling_num)
Inputs: LINEDEV linedev • GlobalCall line device handle
char *calling_num • calling phone number string
Returns: 0 if successful
<0 if failure
Includes: gclib.h
gcerr.h
Category: optional feature
Mode: synchronous
Technology: ■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit
■ Analog

■ Description

The **gc_SetCallingNum( )** function sets default calling party number associated with the specific line device. The calling party number ends with '\0'. The calling party number may also be set using the **gc_SetParm( )** function.

Parameter	Description
linedev:	GlobalCall line device handle
calling_num:	phone number of the calling party (ASCII string format)

Termination Event: None.

■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
```

```
int set_calling_num(LINEDEV ldev)
{
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */

   /* Set up the calling party number on the line device */
   if (gc_SetCallingNum(ldev, "2019933000") != GC_SUCCESS) {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               ldev, gc_error, msg);
      return(gc_error);
   }
   /*
    * Application can proceed to make a call, and the calling party
    * number will be as set above.  It may be changed later, if
    * necessary.
    */
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_MakeCall( )**

Name: int gc_SetChanState(linedev, chanstate, mode)

Inputs:
- LINEDEV linedev — • GlobalCall line device handle
- int chanstate — • channel service state
- unsigned long mode — • async or sync

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: optional feature

Mode: asynchronous or synchronous

Technology: ■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit
❑ Analog

■ Description

The **gc_SetChanState( )** function changes the maintenance state of the indicated channel. When power is initially applied, all channels are placed in the In Service state.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
linedev:	GlobalCall line device handle
chanstate:	service state of line. Possible values are: In Service, Maintenance, and Out of Service, see *Table 34*.
mode:	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

Table 34. Service States

Type	Description
GCLS_INSERVICE	Inform driver that host is ready to receive and send a message.
GCLS_MAINTENANCE	Inform host that normal outbound traffic is not allowed, and that only inbound test calls can be made.
GCLS_OUT_OF_SERVICE	Inform driver that host is not ready to receive or send messages. See the *GlobalCall Technology User's Guide* for your network interface for inbound and outbound requests that will be rejected.

Termination Event: In the asynchronous mode, if the request for a change of channel state is accepted, a GCEV_SETCHANSTATE event is sent to the application; otherwise, a GCEV_TASKFAIL event is sent.

■ Cautions

This function should only be invoked while in the Null state.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

/* Assume following was done:
 *  IF not in the Null state, THEN
 *    issue gc_DropCall() function (if needed) and then the
 *    gc_ReleaseCall() function.
 */
int set_channel_InService(LINEDEV ldev)
{
   int            state;            /* State to which channel has to be set */

   int            cclibid;         /* cclib id for gc_ErrorValue() */
   int            gc_error;        /* GlobalCall error code */
   long           cc_error;        /* Call Control Library error code */
   char           *msg;            /* points to the error message string */

   /*
```

```
    * Set channel to "INSERVICE" state
    */
   state = GCLS_INSERVICE;          /* constant describing channel state */
   if (gc_SetChanState(ldev, state, EV_SYNC) != GC_SUCCESS) {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               ldev, gc_error, msg);
      return(gc_error);
   }

   /*
    * Application can change state again when necessary.
    */
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure or if the GCEV_TASKFAIL event is received, use **gc_ErrorValue( )** or **gc_ResultValue( )**, respectively, and the **gc_ResultMsg( )** function as described in *section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_WaitCall( )**

Name:	int gc_SetEvtMsk(linedev, bitmask, action)	
Inputs:	LINEDEV linedev	• GlobalCall line device handle
	unsigned long bitmask	• bitmask or events
	int action	• action to be taken on the mask bit
Returns:	0 if successful <0 if failure	
Includes:	gclib.h gcerr.h	
Category:	system control and tools	
Mode:	synchronous	
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ■ Analog	

■ Description

The **gc_SetEvtMsk( )** function sets the event mask associated with the specified line device. If an event **bitmask** parameter is cleared, the event will be disabled and will **not** be sent to the application. The default is to enable all events.

The **linedev** parameter may represent a network interface trunk or an individual channel, e.g., a time slot. See the *GlobalCall Technology User's Guide* for your network interface to determine the level of the event masks needed.

Parameter	Description
linedev:	GlobalCall line device handle
bitmask:	specifies the events to be enabled or disabled by setting the bitmask. Multiple transition events may be enabled or disabled with one function call if the bitmask values are bitwise ORed. Possible **bitmask** values are listed in *Table 35*.
action:	application may either set or reset the mask bit(s) as specified in **bitmask**. Possible actions are: • **GCACT_SETMSK:** Enables notification of events specified in **bitmask** parameter and disables notification of any event not specified. • **GCACT_ADDMSK:** Adds notification of events specified in **bitmask** parameter to previously enabled

Parameter	Description

events.

- **GCACT_SUBMSK:** Disables notification of events specified in **bitmask** parameter.

Table 35. *bitmask* Parameter Values

Type	Description
GCMSK_ALERTING	Set mask for alerting event GCEV_ALERTING (default: enabled).
GCMSK_BLOCKED	Set mask for GCEV_UNBLOCKED event (default: enabled).
GCMSK_UNBLOCKED	Set mask for GCEV_UNBLOCKED event (default: enabled).
GCMSK_PROCEEDING	(ISDN only) Set mask for proceeding event GCEV_PROCEEDING (default: enabled).
GCMSK_PROC_SEND	(ISDN only) Set mask (enable) to allow application to send the Proceeding message or clear mask (disable) to have this handled automatically (default: disabled - message automatically sent by firmware).
GCMSK_PROGRESS	(ISDN only) Set mask for call progress event GCEV_PROGRESS (default: enabled).
GCMSK_SETUP_ACK	(ISDN only) Set mask to report (enabled) or to not report (disabled) the incoming "SETUP_ACK" message (default: disabled)

The GCEV_BLOCKED and GCEV_UNBLOCKED events are maskable on a line device representing a trunk or a time slot level. The application may disable (mask) the event on any line device so that the event is not sent to that line device. For example, when a trunk alarm occurs, this alarm is reported via the GCEV_BLOCKED event. If the application has not masked (disabled) this GCEV_BLOCKED event on some or all of the opened time-slot level line devices on the trunk, the GCEV_BLOCKED event will be sent to each of the line devices

on the trunk. Also, if this GCEV_BLOCKED event is not disabled on the board-level line device, the alarm is sent to the board.

The GCEV_ALERTING event is maskable as described above except that this event is always call related and always associated with a time-slot level line device. The time-slot level line device should be passed to the **gc_SetEvtMsk( )** function.

See the *GlobalCall Technology User's Guide* for your network interface for additional details.

Termination Event: None.

■ Cautions

When using the ISDN call control library, if the line device represents an ISDN time slot, setting the mask for an event on any time slot results in setting the mask to the same value for all time slots on the same trunk. The ISDN call control library treats the line device as if it were at board level, thus setting the mask for all time slot level line devices on a trunk when any line device mask is set.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

/*
 * Assume the following has been done:
 *    1. The line device has been opened.
 */
int set_event_mask(LINEDEV ldev)
{
   int           cclibid;         /* cclib id for gc_ErrorValue() */
   int           gc_error;        /* GlobalCall error code */
   long          cc_error;        /* Call Control Library error code */
   char          *msg;            /* points to the error message string */

   /*
    * Set the event blocked and unblocked event masks to enable
    * for this application.
    */
   /*
    * Enable the Blocked and Unblocked events.
    */
   if (gc_SetEvtMsk(ldev, (GCMSK_BLOCKED | GCMSK_UNBLOCKED), GCACT_ADDMSK)
       != GC_SUCCESS) {
      /* process error return as shown */
```

```
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               ldev, gc_error, msg);
      return(gc_error);
   }

   /*
    * Proceed to generate or accept calls on this line device.
    */

   /*
    * Now disable notification of Blocked and Unblocked events,
    * and enable notification of Alerting event, without
    * affecting any other event masks which may have been set.
    */
   if (gc_SetEvtMsk(ldev, (GCMSK_BLOCKED | GCMSK_UNBLOCKED), GCACT_SUBMSK)
      != GC_SUCCESS) {
      /* Process error */
   }
   if (gc_SetEvtMsk(ldev, GCMSK_ALERTING, GCACT_ADDMSK) != GC_SUCCESS) {
      /* Process error */
   }
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_SetParm( )**

Name: int gc_SetInfoElem(linedev, iep)

Inputs:
- LINEDEV linedev — • D channel GlobalCall line device handle
- GC_IE_BLK *iep — • pointer to information element (IE) block

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h
gcisdn.h

Category: interface specific

Mode: synchronous

Technology: ■ ISDN PRI ❑ E-1 CAS ❑ T-1 robbed bit
❑ Analog

■ Description

The **gc_SetInfoElem( )** function allows applications to set an additional information element in the next outbound ISDN message on a specific D channel. This and the facility functions are useful tools for users who wish to use ISDN flexibility and capabilities. A typical application for the **gc_SetInfoElem( )** function is inserting user-to-user information elements in outbound messages.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
linedev:	GlobalCall line device handle
iep:	pointer to the starting address of the information element data structure, see *paragraph 5.2. GC_IE_BLK* for data structure details

Termination Event: None.

■ Cautions

The **gc_SetInfoElem( )** function must be used just prior to calling a function that sends an ISDN message. The information elements specified by the **gc_SetInfoElem( )** function are applicable only to the next outbound ISDN message.

The line device number in the parameter must match the line device number in the function call that sends the ISDN message.

If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

■ Example

```
/*
 * Assume the following has been done:
 * 1. device has been opened (e.g. :N_dtiB1T1:P_isdn,
 *    :N_dtiB1T2:P_isdn, etc...)
 */


#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>

/*
 * the following info elem block structure can be passed to the function.
 *    IE_BLK ie;
 *    ie.length = 0x08;       ===> Length of the info elem block.
 *    ie.data[0] = 0x7e;      ===> User-User info elem id.
 *    ie.data[1] = 0x06;      ===> Length of the info elem.
 *    ie.data[2] = 0x08;      ===> Protocol Discriminator.
 *    ie.data[3] = 0x41;      ===> the following is the message.
 *    ie.data[4] = 0x42;
 *    ie.data[5] = 0x43;
 *    ie.data[6] = 0x44;
 *    ie.data[7] = 0x45;
 */

int set_info_element(LINEDEV ddd, GC_IE_BLK *ie_blkp)
{
  int     gc_err;          /* GlobalCall Error Code */
  int     cclibid;         /* Call Control library ID */
  long    cclib_err;       /* Call Control Error Code */
  char    *msg;            /* Error Message */

  if(gc_SetInfoElem(ddd, ie_blkp) != GC_SUCCESS) {
    gc_ErrorValue(&gc_err, &cclibid, &cclib_err);
    gc_ResultMsg(cclibid, cclib_err, &msg);
```

```
    printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
             ddd, gc_err, msg);
    return(cclib_err);
  }

  return(0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_SetParm( )**

Name: int gc_SetParm(linedev, parm_id, value)

Inputs:
LINEDEV linedev	• GlobalCall line device handle
int parm_id	• parameter ID
GC_PARM value	• parameter value

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h
gcisdn.h (for applications that use ISDN symbols)

Category: system control and tools

Mode: synchronous

Technology: ■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit
❑ Analog

■ Description

The **gc_SetParm( )** function sets the default parameters and all channel information associated with the specific line device.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
linedev:	GlobalCall line device handle
parm_id:	The parameter ID definitions are listed in *Table 36. Parameter Descriptions, gc_GetParm( ) and gc_SetParm( ).* The "Level" column lists whether the parameter is a channel level parameter or a trunk level parameter. To set a trunk level parameter, the **linedev** parameter must be the device ID associated with a network interface trunk; see *paragraph 5.5. GC_PARM* for data structure details.
value:	value selected for parameter being set

Table 36. Parameter Descriptions, gc_GetParm( ) and gc_SetParm( )

Parameter†	Level	Description
GCPR_CALLINGPARTY	chan	Calling party number (pointer to null-terminated ASCII string) (possible values are the existing GTD identification numbers). Use **paddress** field of GC_PARM.
E-1 CAS Parameters:		
GCPR_LOADTONES	chan	Load tones flag enables or disables downloading of tones when a voice resource is attached (possible values: GCPV_ENABLE, GCPV_DISABLE; default: enabled). Use **shortvalue** field of GC_PARM.
ISDN Parameters‡:		
BC_INFO_MODE	chan	Bearer channel information transfer mode
BC_XFER_CAP	chan	Bearer channel information transfer capacity
BC_XFER_MODE	chan	Bearer channel information transfer mode
BC_XFER_RATE	chan	Bearer channel information transfer rate
USRINFO_LAYER1_ PROTOCOL	chan	Layer 1 protocol to use on bearer channel
USR_RATE	chan	User rate to use on bearer channel (layer 1 rate)
CALLED_NUM_TYPE	chan	Called party number type
CALLED_NUM_PLAN	chan	Called party number plan
CALLING_NUM_TYPE	chan	Calling party number type
CALLING_NUM_PLAN	chan	Calling party number plan

Parameter†	Level	Description
CALLING_ PRESENTATION	chan	Calling presentation indicator
CALLING_SCREENING	chan	Calling screening indicator field
GCPR_MINDIGITS	trunk	Sets minimum number of DDI digits to collect prior to terminating **gc_WaitCall( ).** GCPR_MINDIGITS may be set using the **gc_SetParm( )** function. This parameter value cannot be retrieved using the **gc_GetParm( )** function.

† = See the *GlobalCall Technology User's Guide* for your network interface to determine applicable parameters.
‡ = All ISDN parameters use the **intvalue** field of GC_PARM.

Termination Event: None.

■ Cautions

None.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>

int set_parm(ldev)
{
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */
   GC_PARM        gc_parm;          /* parm values */
   /*
    * Disable downloading tones to firmware. This is to prevent GlobalCall
    * from overwriting tones which the application has set up
    */
   gc_parm.shortvalue = GCPV_DISABLE;
   if ( gc_SetParm(ldev, GCPR_LOADTONES, gc_parm) != GC_SUCCESS) {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               ldev, gc_error, msg);
      return(gc_error);
```

```
    }
    return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_GetParm( )**

Name: int gc_SetUsrAttr(linedev, usrattr)

Inputs: LINEDEV linedev • GlobalCall line device handle
void *usrattr • user attribute

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: system control and tools

Mode: synchronous

Technology: ■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit
■ Analog

■ Description

The **gc_SetUsrAttr( )** function sets an attribute defined by the user. Examples of using **usrattr** include using it as a pointer to a data structure associated with a line device or an index to an array. The data structure may contain user information such as the current call state, line device identification, etc. The attribute number is retrieved using the **gc_GetUsrAttr( )** function.

Parameter	Description
linedev:	GlobalCall line device handle
usrattr:	user defined attribute. Applications can recall this number by calling **gc_GetUsrAttr( ).**

Termination Event: None

■ Cautions

None

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
```

```
#include <gcerr.h>

#define MAXCHAN   30                /* max. number of channels in system */
/*
 * Data structure which stores all information for each line
 */
struct linebag {
   LINEDEV  ldev;                   /* GlobalCall line device handle */
   CRN      crn;                    /* GlobalCall API call handle */
   int      state;                  /* state of first layer state machine */
} port[MAXCHAN+1];

/*
 * Associates port_num with ldev for later use
 * by other procedures - will save table searches
 * for the port_num corresponding to ldev
 */
int set_usrattr(LINEDEV ldev, int port_num)
{
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */

   /*
    * Assuming that a line device is opened already and
    * that its ID is ldev, let us store a meaningful number
    * for this ldev as an attribute for this ldev set by user
    */
   if (gc_SetUsrAttr(ldev, (void *) port_num) != GC_SUCCESS) {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
               ldev, gc_error, msg);
       return(gc_error);
   }
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_GetUsrAttr( )**
- **gc_OpenEx( )**

Name:	int gc_SndMsg(linedev, crn, msg_type, sndmsgptr)	
Inputs:	LINEDEV linedev	• line device number for the B channel
	CRN crn	• call reference number
	int msg_type	• ISDN message type
	GC_IE_BLK *sndmsgptr	• pointer to the Information Element (IE) block
Returns:	0 if successful <0 if failure	
Includes:	gclib.h gcerr.h gcisdn.h	
Category:	interface specific	
Mode:	synchronous	
Technology:	■ ISDN PRI ❑ E-1 CAS ❑ T-1 robbed bit ❑ Analog	

■ Description

The **gc_SndMsg( )** function sends non-call state related ISDN message to the network over the D channel while a call exists. The data is sent transparently over the D channel data link with LAPD protocol.

NOTE: The message must be sent over a channel that has a CRN assigned to it.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
linedev:	line device number for the time slot level line device (the B channel)
crn:	Call Reference Number. Each call needs a CRN.
msg_type:	specifies the type of message to be sent, see the appropriate *GlobalCall Technology User's Guide* for details.
sndmsgptr:	pointer to the buffer that contains the IEs to be sent in the message; see *paragraph 5.2. GC_IE_BLK* for data structure details.

Termination Event: None.

■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

For some call control libraries (e.g., ISDN library), if an invalid parameter is used for a **gc_SndMsg( )** call, then the invalid parameter is ignored, processing continues and the function terminates normally.

■ Example

```
/*
 * Assume the following has been done:
 * 1. device has been opened (e.g. :N_dtiB1T1:P_isdn,
 *    :N_dtiB1T2:P_isdn, etc...)
 * 2. gc_WaitCall() has been issued to wait for a call.
 * 3. gc_MetaEvent() or gc_GetMetaEventEx() (Windows NT) has been
 *    called to convert the event into metaevent.
 * 4. a GCEV_OFFERED has been detected.
 * 5. a call has been established.
 */


#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>

/*
 * the following info elem block structure can be passed to the function.
 *    IE_BLK ie;
 *    ie.length = 0x08;       ===> Length of the info elem block.
 *    ie.data[0] = 0x7e;      ===> User-User Info elem id.
 *    ie.data[1] = 0x06;      ===> Length of the info elem.
 *    ie.data[2] = 0x08;      ===> Protocol Discriminator.
 *    ie.data[3] = 0x41;      ===> the following is the message.
 *    ie.data[4] = 0x42;
 *    ie.data[5] = 0x43;
 *    ie.data[6] = 0x44;
 *    ie.data[7] = 0x45;
 */

int send_message(CRN crn, int msg_type, GC_IE_BLK *sndmsgp)
{
  LINEDEV  ddd;             /* Line device */
  int      gc_err;          /* GlobalCall Error Code */
  int      cclibid;         /* Call Control library ID */
  long     cclib_err;       /* Call Control Error Code */
  char     *msg;            /* Error Message */
```

```
  if(gc_CRN2LineDev(crn, &ddd) != GC_SUCCESS) {
    gc_ErrorValue(&gc_err, &cclibid, &cclib_err);
    gc_ResultMsg(cclibid, cclib_err, &msg);
    printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
             ddd, gc_err, msg);
    return(cclib_err);
  }

  if(gc_SndMsg(ddd, crn, msg_type, sndmsgp) != GC_SUCCESS) {
    gc_ErrorValue(&gc_err, &cclibid, &cclib_err);
    gc_ResultMsg(cclibid, cclib_err, &msg);
    printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
             ddd, gc_err, msg);
    return(cclib_err);
  }

  return(0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_Close( )**

Name:	int gc_Start(startp)
Inputs:	GC_START_STRUCT *startp • reserved for future use
Returns:	0 if successful <0 if failure
Includes:	gclib.h gcerr.h
Category:	System controls and tools
Mode:	synchronous
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ■ Analog

■ Description

The **gc_Start( )** function starts all configured call control libraries. This function MUST be called before any other GlobalCall function is called. The function opens the call control libraries that interface directly to the network interface so that these libraries can be used by the GlobalCall library.

This function returns 0 if all call control libraries have successfully started. Successfully started libraries are available to be used by the GlobalCall functions and are called “available libraries.” Libraries which fail to start are called “failed” libraries.

To avoid link errors in UNIX applications wherein a particular call control library is not required, a library with a minimal set of internal functions is provided. This library is called a “stub” library and it is entered into the list of configured call control libraries recognized by the GlobalCall API. A stub library is not capable of being started and thus does not become available. Th. Non-stub libraries which fail to start are called “failed” libraries.

For UNIX applications, the **gc_Start( )** function must be called from the parent process when creating child processes.

For Windows NT applications, the **gc_Start( )** function must be called from the primary thread when creating multiple threads. The **gc_Stop( )** function must be called from the same thread that issued the **gc_Start( )** call.

Parameter	Description
startp:	reserved for future use: set **startp** to NULL.

Termination Event: None

Use the **gc_CCLibStatusAll( )** function to determine the number and status (started, configured, failed, stub) of all call control libraries.

■ Cautions

This function must be called BEFORE calling other GlobalCall functions and should not be called again until **gc_Stop( )** is called. An error is returned if the **gc_Start( )** function is called more than once without calling the **gc_Stop( )** function.

For UNIX applications, this function must be called from the parent process when creating child processes.

For Windows NT applications, this function must be called from the primary thread when creating multiple threads.

This function automatically calls the **gc_Stop( )** function to stop any library that may be running before starting all libraries.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

int sysinit()
{
   GC_START_STRUCT startp;          /* Structure for gc_Start() */
   int             cclibid;         /* cclib id for gc_ErrorValue() */
   int             gc_error;        /* GlobalCall error code */
   long            cc_error;        /* Call Control Library error code */
   char            *msg;            /* points to the error message string */

   /* Open all necessary vox/log files */

   /* Next issue a gc_Start() Call */
```

```
   memset(&startp, '\0', sizeof(GC_START_STRUCT));
   if ( gc_Start( &startp ) != GC_SUCCESS )    {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("Error in gc_Start ErrorValue: %d - %s\n",
               gc_error, msg);
      return(gc_error);
   }

   /* Next open the GlobalCall Line Devices */

   return(0);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

A **gc_Start( )** function can fail for multiple, simultaneous causes. For example, both of the following failure conditions might be present and therefore must be rectified:

- EGC_CCLIBSTART indicates that at least one call control library failed to start.
- EGC_ALARMDBINIT indicates that the alarm database failed to initialize, probably due to insufficient dynamic memory.

If this function returns a -1, then all configured libraries did not start successfully.

■ See Also

- **gc_CCLibStatusAll( )**
- **gc_Stop( )**

Name: int gc_StartTrace(linedev, filename)

Inputs: LINEDEV linedev • GlobalCall line device handle
char *filename • file name for trace

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: interface specific

Mode: asynchronous

Technology: ■ ISDN PRI ❑ E-1 CAS ❑ T-1 robbed bit
❑ Analog

■ Description

The **gc_StartTrace( )** function instructs the firmware to trace and place results in shared RAM. This function opens a file under the **filename** parameter and saves the results to this file. This function allows the application to trace ISDN messages on the specified D channel. The saved trace file is interpreted off line by the PRITRACE utility program supplied with the software package. The trace continues until a **gc_StopTrace( )** function is issued.

NOTE: The **linedev** parameter must use the line device number for the D channel board.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
linedev:	GlobalCall line device handle of D channel board.
filename:	specifies file name for the trace.

Termination Event: None. The trace initiated by this function continues until a **gc_StopTrace( )** function is issued for the line device.

■ Cautions

If the **gc_StartTrace( )** function was issued, the application should call the **gc_StopTrace( )** function before calling the **gc_Close( )** function for that line device.

When using the **gc_StartTrace( )** function, only one board can be traced at a time. When using UNIX or Windows NT single process programming, an error is returned if the **gc_StartTrace( )** function is issued when a trace is currently running on another board.

If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

LINEDEV bdev;                       /* board level device number */
int     parm_id;                    /* parameter id */
int     rc;                         /* Return code */
int     value;                      /* value to be for specified parameter */
char    *filename;                  /* file name for the trace */
int     cclibid;                    /* cclib id for gc_ErrorValue() */
int     gc_error;                   /* GlobalCall error code */
long    cc_error;                   /* Call Control Library error code */
char    *msg;                       /* points to the error message string */

main()
{
   if(gc_Open(&bdev, "dtiB1", 0) != GC_SUCCESS) {
      gc_ErrorValue( &gc_error, &cclibid, &cclib_err);
      gc_ResultMsg( cclibid, cclib_error, &msg);
      printf ("Error: gc_Open, ErrorValue: %d - %s\n",
               cclib_err, msg);
       return(gclib_err);
   }

   /* Only one D channel can be traced at any given time  */
   .
   .
   .

   filename="/tmp/trace.log";
   rc = gc_StartTrace(bdev, filename);
   if (rc != GC_SUCCESS) {
      printf("Error in gc_StartTrace, rc = %x\n", rc);
   } else {
      /* continue */
```

```
    }
    .
    .
    .
}
```

Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

See Also

- **gc_StopTrace( )**

Name:	int gc_Stop(void)
Inputs:	none
Returns:	0 if successful <0 if failure
Includes:	gclib.h gcerr.h
Category:	System controls and tools
Mode:	synchronous
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit ■ Analog

■ Description

The **gc_Stop( )** function stops all configured call control libraries started and cleans-up the GlobalCall database. This function MUST be the last GlobalCall function called before exiting the application or issuing another **gc_Start( )** function.

For UNIX applications, this function must be called from the parent process when child processes are used.

For Windows NT applications, the **gc_Stop( )** function must be called from the same thread that issued the **gc_Start( )** call.

Termination Event: None

■ Cautions

This function must be called before exiting the application. If this function fails, exit your application before issuing another **gc_Start( )** function. This function must be called from the parent process when child processes are used.

All open devices should be closed before issuing a **gc_Stop( )** function.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
```

```
#include <stdlib.h>
#include <gclib.h>
#include <gcerr.h>

#define MAXCHAN 30                  /* Total Number of channels opened */

LINEDEV port[MAXCHAN + 1];          /* Array of line devices previously opened */

void sysexit( int exit_code )
{
   int            port_num;         /* Index used for port[] */
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */

   /* First close all the handles for the opened boards */

   /* Now close all the open GlobalCall devices */
   for (port_num = 1; port_num <= MAXCHAN; port_num++ )  {
      if (gc_Close(port[port_num].ldev) != GC_SUCCESS ) {
         /* Process error return from gc_Close() */
      }
   }

   /* Issue gc_Stop() Next */
   if (gc_Stop() != GC_SUCCESS ) {
      /* process error return as shown */
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
      gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
      printf ("gc_Stop returns error ErrorValue: %d - %s\n",
               gc_error, msg);
   }

   /* Close all open file handles corresponding to recorded files  and exit */
   exit(exit_code);
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_Start( )**

Name: int gc_StopTrace(linedev)
Inputs: LINEDEV linedev • GlobalCall line device handle
Returns: 0 if successful
<0 if failure
Includes: gclib.h
gcerr.h
Category: interface specific
Mode: synchronous
Technology: ■ ISDN PRI ❑ E-1 CAS ❑ T-1 robbed bit
❑ Analog

■ Description

The **gc_StopTrace( )** function stops the trace that was started using the **gc_StartTrace( )** function.

Parameter	Description
linedev:	GlobalCall line device handle of D channel board.

Termination Event: None

■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value EGC_UNSUPPORTED will be the GlobalCall value returned when the **gc_ErrorValue( )** function is used to retrieve the error code.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

LINEDEV         bdev;                   /* board level device number */
int             parm_id;                /* parameter id */
int             rc;                     /* Return code */
int             value;                  /* value to be for the specified parameter */
int             D_CH_hdl;               /* identify D channel to be traced */
char            *filename;              /* file name for the trace */
```

```
main()
{
/* Only one D channel can be traced at any given time. */
   .
   .
   .
   rc = gc_StopTrace(bdev);
   if (rc != GC_SUCCESS) {
      printf("Error in gc_StopTrace, rc = %x\n", rc);
   } else {
      /* Process event */
   }
   .
   .
   .
}
```

■ Errors

If this function returns a <0 to indicate failure, use the **gc_ErrorValue( )** and **gc_ResultMsg( )** functions as described in *Section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_StartTrace( )**

Name:	int gc_WaitCall(linedev, crnp, waitcallp, timeout, mode)	
Inputs:	LINEDEV linedev	• GlobalCall line device handle
	CRN *crnp	• pointer to CRN
	GC_WAITCALL_BLK *waitcallp	• reserved for future use
	int timeout	• time-out
	unsigned long mode	• async or sync
Returns:	0 if successful	
	>0, if failure	
Includes:	gclib.h	
	gcerr.h	
Category:	basic call control	
Mode:	asynchronous or synchronous	
Technology:	■ ISDN PRI ■ E-1 CAS ■ T-1 robbed bit	
	■ Analog	

■ Description

The **gc_WaitCall( )** function sets up conditions for processing inbound calls. The **gc_WaitCall( )** function unblocks the time slot (if the technology and the line conditions permit unblocking the line) and enables notification of inbound calls:

- For E-1 CAS and T-1 robbed bit applications, the line will be set to IDLE after the first call to a **gc_WaitCall( )** function.
- Analog technology does not provide a means to physically block or unblock an analog line.
- For ISDN applications, the state will be set to NULL after the call to a **gc_WaitCall( )** function.

In the asynchronous mode, after the **gc_WaitCall( )** function was successfully called, the **gc_ReleaseCall( )** function will not block the incoming notification. Therefore, it is only necessary to call a **gc_WaitCall( )** function once**.** A subsequent usage of the **gc_WaitCall( )** function in the asynchronous mode has no additional effect. Also, the call reference parameter is not used in this function call. The application must retrieve the CRN from the metaevent structure returned when the call notification event (GCEV_OFFERED) arrives.

In the synchronous mode, notification of the next inbound call is blocked until the next **gc_WaitCall( )** function is issued. If an inbound call arrives between the **gc_ReleaseCall( )** and **gc_WaitCall( )** functions, the call will be pending until **gc_WaitCall( )** function is reissued, at which point the application will be notified.

When called in the synchronous mode, the **crnp** parameter is assigned when the **gc_WaitCall( )** function terminates. If the **gc_WaitCall( )** function fails, the call (and thus the CRN) will be released automatically.

Refer also to the appropriate *GlobalCall Technology User's Guide* for technology specific information.

Parameter	Description
linedev:	GlobalCall line device handle
crnp:	pointer to the CRN. The **crnp** parameter must be of a global and non-temporary type. The **crnp** parameter is used only in the synchronous mode. For the asynchronous mode, this parameter must be set to null. When the GCEV_OFFERED event is received, the CRN can be retrieved.
waitcallp:	not used in this release. Set to NULL.
timeout:	used only in synchronous mode, ignored in asynchronous mode - specifies the interval (in seconds) to wait for the call. When the timeout expires, the function will return -1 and the call will remain in the Null state. The error value is set to EGC_TIMEOUT. If the timeout is 0 and no inbound call is pending, the function returns -1 with an EGC_TIMEOUT error value. In synchronous mode, another **gc_WaitCall( )** function may be issued immediately without issuing a **gc_DropCall( )** or **gc_ReleaseCall( )** function.
mode:	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

Termination Event: None

In the asynchronous mode, the **gc_WaitCall( )** function does not return an event and is assumed to have successfully completed when issued. The unsolicited event GCEV_OFFERED may be received later.

■ Cautions

The application should always call a **gc_ReleaseCall( )** function to release the CRN after the termination of a connection. Failure to do so may cause memory problems due to memory being allocated and not being released.

In the asynchronous mode, the CRN will not be available until an inbound call has arrived (i.e., GCEV_OFFERED received).

For both the asynchronous and the synchronous modes, any active **gc_WaitCall( )** function can be stopped by using the **gc_ResetLineDev( )** function. When the **gc_ResetLineDev( )** function completes, the application must reissue the **gc_WaitCall( )** function to be able to receive incoming calls.

■ Example

```
#include <windows.h>            /* For Windows NT applications only */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>

#define MAXCHAN   30            /* max. number of channels in system */
/*
 * Data structure which stores all information for each line
 */
struct linebag {
   LINEDEV  ldev;               /* line device handle */
   CRN      crn;                /* GlobalCall API call handle */
   int      state;              /* state of first layer state machine */
} port[MAXCHAN+1];
struct linebag *pline;          /* pointer to access line device */

/*
 * Assume the following has been done:
 *    1. Open line devices for each time slot on dtiB1.
 *    2. Each Line Device ID is stored in linebag structure, 'port'.
 */
int wait_call(int port_num)
{
   int            cclibid;          /* cclib id for gc_ErrorValue() */
   int            gc_error;         /* GlobalCall error code */
   long           cc_error;         /* Call Control Library error code */
   char           *msg;             /* points to the error message string */

   /* Find info for this time slot, specified by 'port_num' */
   pline = port + port_num;
```

```
   /*
    * Wait for a call, with 0 timeout.
    */
   if (pline->state == GCST_NULL) {
      if (gc_WaitCall(pline->ldev, NULL, NULL, 0, EV_ASYNC) != GC_SUCCESS) {
         /* process error return as shown */
         gc_ErrorValue( &gc_error, &cclibid, &cc_error);
         gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
         printf ("Error on Device handle: 0x%lx, ErrorValue: %d - %s\n",
                  pline -> ldev, gc_error, msg);
         return(gc_error);
      }
   }
   /*
    * GCEV_OFFERED event will indicate incoming call has arrived.
    */
   return (0);
}
```

■ Errors

If this function returns a <0 to indicate failure or if the GCEV_TASKFAIL event is received, use **gc_ErrorValue( )** or **gc_ResultValue( )**, respectively, and the **gc_ResultMsg( )** function as described in *section 3.11. Error Handling* to retrieve the reason for the error. All GlobalCall error codes are defined in the *gcerr.h* file, see listing in *Appendix C*.

■ See Also

- **gc_DropCall( )**
- **gc_MakeCall( )**
- **gc_ReleaseCall( )**
- **gc_ResetLineDev( )**

7. GlobalCall Demo Programs

GlobalCall UNIX and Windows NT inbound and outbound demonstration programs illustrating the application of GlobalCall functions are described in this chapter in terms of:

- an overview of the GlobalCall demo programs
- the physical connection required to run these demo programs
- preparing to run the GlobalCall demo programs and
- running the GlobalCall demo programs

7.1. Demo Programs for UNIX

The following paragraphs describe analog technology and E-1/T-1 technology demonstrations that run on a UNIX platform.

The demo programs use user-modifiable configuration files that define the protocol to be run on each channel and the voice and/or network (E-1/T-1) resources to be used. Separate configuration files can be defined for inbound (*gcin.cfg*) calls, outbound (*gcout.cfg*) calls and for analog (*gcanalog.cfg*) technology only calls. In addition to compilable files, executable demo files using sample configuration files similar to those described in this chapter are stored in the Dialogic */usr/dialogic/gc_demos* directory.

The GlobalCall demonstration programs are:

- *inbound:* demonstrates operation of the GlobalCall API for handling inbound calls
- *outbound*: demonstrates operation of the GlobalCall API for handling outbound calls

The demonstration programs operate independent of each other. Each program implements a double layer state machine based on the GlobalCall API. The first layer deals with the GlobalCall call establishment and termination processes. This layer includes the following states:

- ST_BLOCKED,
- ST_NULL,
- ST_OFFERED,
- ST_TALK,
- ST_CLOSING,
- ST_DISCONNECTED and
- ST_IDLE.

The second layer deals with events that can occur during a conversation (the ST_TALK state) and includes the following states for the inbound program:

- WELCOME,
- RECORD,
- GOODBYE,
- GETDIGIT,
- INVALID,
- PLAYBACK and
- STOPPING

The outbound program uses only the WELCOME and the STOPPING states in the second layer.

Figure 7. UNIX Demo Program States illustrates the structure of the GlobalCall demo programs.

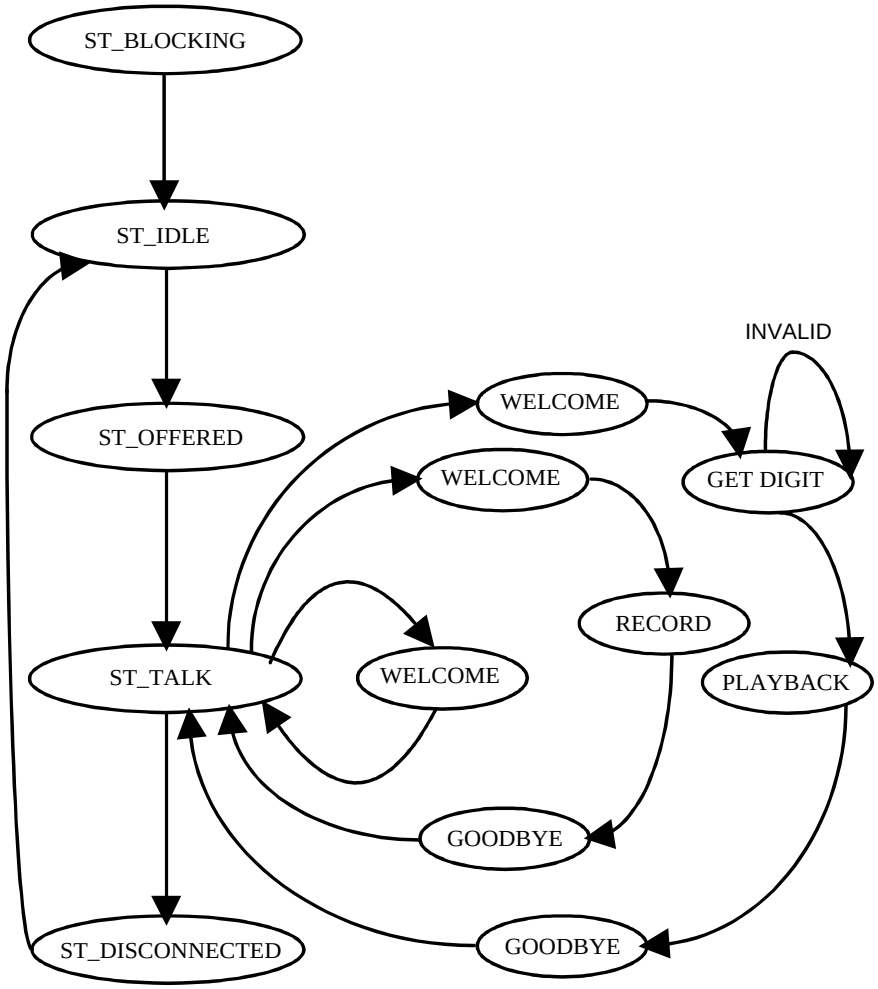

Figure 7. UNIX Demo Program States

Start the GlobalCall demo programs from the command line. Select the parameters and options you wish to use by typing the parameter value or option details after the appropriate option switch (see *Section 7.1.4. Running the UNIX Demo Program*.

A LINEBAG data structure contained in the demo software holds the state of each line device. The demo programs assume that voice channel *n* is routed to DTI time slot *n*. Unless you use the *-n* switch to specify a different number, the programs will open as many devices as there are voice channels.

The GlobalCall distribution diskettes contain all the demonstration program files. These files are installed on your system in the */usr/dialogic/gc_demos* installation directory when you install the GlobalCall software. The source code for the demonstration programs is written in the C programming language.

7.1.1. Physical Connections for the UNIX Demo

To run the GlobalCall Demo programs, you need one or more of the following:

- a connection to the network (analog loop start, E-1 CAS/T-1 robbed bit or ISDN)
- an E-1, T-1 and/or ISDN simulator
- analog loop start simulator or Dialogic PromptMaster development tool

You may make this connection either before or after installing the GlobalCall software.

7.1.2. Before Running the UNIX Demo Programs

GlobalCall software must be installed to run the GlobalCall Demo programs. To run the included executable demo programs or your compiled demos, see *Paragraph 7.1.4. Running the UNIX Demo Program*

To recompile the demonstration programs using configuration files you created, perform the following:

NOTE: The ANAPI, ICAPI and ISDN call control libraries (or the equivalent stub library) must be installed. Change the *makefile* to include the appropriate stub libraries to match the system configuration.

- while logged on to the system with root privileges, change to the */usr/dialogic/gc_demos* installation directory.
- to compile the inbound program, type:

  make inbound <Enter>

- to compile the outbound program, type:

  make outbound <Enter>

NOTE: A protocol package must be installed on the system, and the *makefile* must use an installed protocol. Initially, the protocols specified in the *makefile* are the *ar_r2_i* and the *ar_r2_o*. Be sure to modify the *makefile* to use the protocol(s) installed on the system.

7.1.3. Demo Configuration Files

The executable demo programs stored in the */usr/dialogic/gc_demos* directory were compiled using sample ASCII configuration files such as shown in:

- *Figure 8. Inbound (gcin_r2is.cfg) Configuration Sample File* ,
- *Figure 9. Outbound (gcout_anis.cfg) Configuration Sample File* and
- *Figure 10. Analog (gcanalog.cfg) Technology Configuration Sample File* .

You can use these sample configuration files unchanged when you compile your demo program or you can edit them (using a text editor such as the vi editor) to include the protocols and products used by your application.

Each channel can run a different protocol IF the *.prm* file parameters associated with these protocols are compatible. To ensure compatibility, check that the parameters specified in the *.prm* file for a board will work for all protocols that will be run on that board. The parameters in the *.prm* file are downloaded at system initialization, become part of the firmware and cannot be changed by the application.

The protocol and resource information for each channel and the telephone number dialed (up to 24 digits) are defined in these configuration files on a channel by channel basis. The configuration is specified in the following order:

voice channel protocol analog (1=yes,0=no) network (optional) phone number

A digital network interface is not used for an analog call; if specified, the digital interface entry is ignored.

For example, using the following lines taken from *Figure 9. Outbound (gcout_anis.cfg) Configuration Sample File*:

```
dxxxB8C2  ar_r2_o     0           dtiB1T30     4812
dxxxB9C1  na_an_io    1                          11
```

wherein the first line specifies that:

- voice channel 2 on board 8 (dxxxB8C2) will provide the voice resources and will be connected to the digital network interface resource dtiB1T30,
- the Argentina R2 (*ar_r2_o*) outbound protocol will be used,
- a digital network interface is selected; 0 entry equates to a digital network interface,
- time slot 30 on E-1 digital network interface board 1 (dtiB1T30) will be connected to dxxxB8C2 (board 8, voice channel 2) and
- the telephone number to dial is 4812.

wherein the second line specifies that:

- voice channel 1 on board 9 (dxxxB9C1) will provide the voice resources and the analog network interface resource
- the North America analog (*na_an_io*) bi-directional protocol will be used,
- an analog network interface is selected; 1 entry selects analog interface,
- no digital network interface resource is used and
- the telephone number to dial is 11.

The sample inbound configuration file (*gcin_r2is.cfg*) shown in *Figure 8. Inbound (gcin_r2is.cfg) Configuration Sample File*, configures two E-1 spans to handle inbound calls on 60 digital interface channels using the Argentina R2 inbound protocol (*ar_r2_i*) on one span and ISDN protocol on the second span. A voice resource is dedicated to each digital interface.

The outbound configuration file (*gcout_anis.cfg*) shown in *Figure 9. Outbound (gcout_anis.cfg) Configuration Sample File*, configures a single E-1 span to handle outbound calls on 30 digital interface channels using the ISDN protocol with a voice resource dedicated to each digital interface. This file also configures a single channel of a four channel voice board with analog network interfaces to handle outbound calls using the North America analog bidirectional protocol (*na_an_io*).

The analog technology configuration file (*gcanalog.cfg*) shown in *Figure 10. Analog (gcanalog.cfg) Technology Configuration Sample File*, configures a single four channel voice board with analog network interfaces to handle either inbound or outbound calls using the North America analog bidirectional protocol (*na_an_io*).

```
#  Demo configuration file for configuring voice channels, network
#  channels, protocol, analog flag, and phone number
#  This configuration file is for 2 E1 spans (Inbound Config)
#
#  voice   protocol  analog           network       phone
# channel          (1=yes,0=no)      (optional)     number
 dxxxB1C1   ar_r2_i     0              dtiB1T1      1234567
 dxxxB1C2   ar_r2_i     0              dtiB1T2      2345567
 dxxxB1C3   ar_r2_i     0              dtiB1T3      3456567
 dxxxB1C4   ar_r2_i     0              dtiB1T4      4567567
 dxxxB2C1   ar_r2_i     0              dtiB1T5      5678567
 dxxxB2C2   ar_r2_i     0              dtiB1T6      6789567
 dxxxB2C3   ar_r2_i     0              dtiB1T7      7890567
 dxxxB2C4   ar_r2_i     0              dtiB1T8      8901567
 dxxxB3C1   ar_r2_i     0              dtiB1T9      9012567
 dxxxB3C2   ar_r2_i     0              dtiB1T10     1357567
 dxxxB3C3   ar_r2_i     0              dtiB1T11     3579567
 dxxxB3C4   ar_r2_i     0              dtiB1T12     5791567
 dxxxB4C1   ar_r2_i     0              dtiB1T13     7913567
 dxxxB4C2   ar_r2_i     0              dtiB1T14     9135567
 dxxxB4C3   ar_r2_i     0              dtiB1T15     2468567
 dxxxB4C4   ar_r2_i     0              dtiB1T16     4680567
 dxxxB5C1   ar_r2_i     0              dtiB1T17     6802567
 dxxxB5C2   ar_r2_i     0              dtiB1T18     8024567
 dxxxB5C3   ar_r2_i     0              dtiB1T19     2581567
 dxxxB5C4   ar_r2_i     0              dtiB1T20     1234567
 dxxxB6C1   ar_r2_i     0              dtiB1T21     2345567
 dxxxB6C2   ar_r2_i     0              dtiB1T22     3456567
 dxxxB6C3   ar_r2_i     0              dtiB1T23     4567567
 dxxxB6C4   ar_r2_i     0              dtiB1T24     5678567
 dxxxB7C1   ar_r2_i     0              dtiB1T25     6789567
 dxxxB7C2   ar_r2_i     0              dtiB1T26     7890567
 dxxxB7C3   ar_r2_i     0              dtiB1T27     8901567
 dxxxB7C4   ar_r2_i     0              dtiB1T28     9012567
 dxxxB8C1   ar_r2_i     0              dtiB1T29     3691567
 dxxxB8C2   ar_r2_i     0              dtiB1T30     4812567
 dxxxB9C1   isdn        0              dtiB2T1
 dxxxB9C2   isdn        0              dtiB2T2
 dxxxB9C3   isdn        0              dtiB2T3
 dxxxB9C4   isdn        0              dtiB2T4
 dxxxB10C1  isdn        0              dtiB2T5
 dxxxB10C2  isdn        0              dtiB2T6
 dxxxB10C3  isdn        0              dtiB2T7
 dxxxB10C4  isdn        0              dtiB2T8
 dxxxB11C1  isdn        0              dtiB2T9
 dxxxB11C2  isdn        0              dtiB2T10
 dxxxB11C3  isdn        0              dtiB2T11
 dxxxB11C4  isdn        0              dtiB2T12
 dxxxB12C1  isdn        0              dtiB2T13
 dxxxB12C2  isdn        0              dtiB2T14
 dxxxB12C3  isdn        0              dtiB2T15
 dxxxB12C4  isdn        0              dtiB2T16
 dxxxB13C1  isdn        0              dtiB2T17
 dxxxB13C2  isdn        0              dtiB2T18
 dxxxB13C3  isdn        0              dtiB2T19
 dxxxB13C4  isdn        0              dtiB2T20
 dxxxB14C1  isdn        0              dtiB2T21
 dxxxB14C2  isdn        0              dtiB2T22
 dxxxB14C3  isdn        0              dtiB2T23
 dxxxB14C4  isdn        0              dtiB2T24
 dxxxB15C1  isdn        0              dtiB2T25
 dxxxB15C2  isdn        0              dtiB2T26
```

```
dxxxB15C3  isdn       0             dtiB2T27
dxxxB15C4  isdn       0             dtiB2T28
dxxxB16C1  isdn       0             dtiB2T29
dxxxB16C2  isdn       0             dtiB2T30
```

Figure 8. Inbound (gcin_r2is.cfg) Configuration Sample File

```
#  Demo configuration file for configuring voice channels, network
#  channels, protocol, analog flag, and phone number
#  This configuration file is for 1 E1 span and analog (Outbound Config)
#
#  voice   protocol  analog       network      phone
# channel          (1=yes,0=no)  (optional)    number
 dxxxB1C1  isdn     0             dtiB1T1      1234567
 dxxxB1C2  isdn     0             dtiB1T2      2345567
 dxxxB1C3  isdn     0             dtiB1T3      3456567
 dxxxB1C4  isdn     0             dtiB1T4      4567567
 dxxxB2C1  isdn     0             dtiB1T5      5678567
 dxxxB2C2  isdn     0             dtiB1T6      6789567
 dxxxB2C3  isdn     0             dtiB1T7      7890567
 dxxxB2C4  isdn     0             dtiB1T8      8901567
 dxxxB3C1  isdn     0             dtiB1T9      9012567
 dxxxB3C2  isdn     0             dtiB1T10     1357567
 dxxxB3C3  isdn     0             dtiB1T11     3579567
 dxxxB3C4  isdn     0             dtiB1T12     5791567
 dxxxB4C1  isdn     0             dtiB1T13     7913567
 dxxxB4C2  isdn     0             dtiB1T14     9135567
 dxxxB4C3  isdn     0             dtiB1T15     2468567
 dxxxB4C4  isdn     0             dtiB1T16     4680567
 dxxxB5C1  isdn     0             dtiB1T17     6802567
 dxxxB5C2  isdn     0             dtiB1T18     8024567
 dxxxB5C3  isdn     0             dtiB1T19     2581567
 dxxxB5C4  isdn     0             dtiB1T20     1234567
 dxxxB6C1  isdn     0             dtiB1T21     2345567
 dxxxB6C2  isdn     0             dtiB1T22     3456567
 dxxxB6C3  isdn     0             dtiB1T23     4567567
 dxxxB6C4  isdn     0             dtiB1T24     5678567
 dxxxB7C1  isdn     0             dtiB1T25     6789567
 dxxxB7C2  isdn     0             dtiB1T26     7890567
 dxxxB7C3  isdn     0             dtiB1T27     8901567
 dxxxB7C4  isdn     0             dtiB1T28     9012567
 dxxxB8C1  isdn     0             dtiB1T29     3691567
 dxxxB8C2  isdn     0             dtiB1T30     4812567
 dxxxB17C2  na_an_io  1                          101
```

Figure 9. Outbound (gcout_anis.cfg) Configuration Sample File

```
#  This configuration file is for analog calls only
#
# voice    protocol  analog       network    phone
# channel          (1=yes,0=no)  (optional)  number
 dxxxB1C1  na_an_io     1                      11
 dxxxB1C2  na_an_io     1                      12
 dxxxB1C3  na_an_io     1                      13
 dxxxB1C4  na_an_io     1                      14
```

Figure 10. Analog (gcanalog.cfg) Technology Configuration Sample File

7.1.4. Running the UNIX Demo Program

Start either of the demo programs by typing the program name at the command line, followed by the appropriate switch(es). The structure of the demo command is:

inbound *-n<numlines> -f<filename.cfg> -d<numddi>*

outbound *-n<numlines> -f<filename.cfg>*

where:

-n<numlines>	Number of connected lines to use for demo calling (default = 60)
-f<filename>	name of configuration file used (e.g., *gcin.cfg*, *gcout.cfg*, *gcanalog.cfg*, etc.) to setup demo calls.
-d<numddi>	DDI (Direct Dialing In) number threshold for call rejection. Incoming calls with a number of DDI digits greater than the number specified will be rejected.

Note that switch “d” is invalid for outbound calls.

NOTE: A protocol package must be installed on the system prior to running the demo programs. The configuration file must specify an installed protocol. *Refer to the GlobalCall Technology User’s Guide* for your technology for information on installing protocols.

NOTE: A protocol must be installed on the system. Before running a demo program that uses a T-1 robbed bit protocol, disable the DTI Wait Call function in the *icapi.cfg* file. See the *icapi.cfg File paragraph* in the *GlobalCall E-1/T-1 Technology User's Guide* for details.

For example: *inbound -n42 -d6 -fgcin_r2is.cfg* <Enter>

is the command you would type on the command line to handle 42 inbound digital network calls with a maximum of 6 DDI digits on the first 42 E-1 channels defined in the configuration file shown in *Figure 8. Inbound (gcin_r2is.cfg) Configuration Sample File*.

For example: *outbound -n31 -fgcout_anis.cfg* <Enter>

is the command you would type on the command line to handle 30 outbound digital network calls on the first 30 E-1 channels defined in the configuration file shown in *Figure 9. Outbound (gcout_anis.cfg) Configuration Sample File* and 1 outbound analog network call on the 4 analog channels of virtual board 17.

For example: *inbound -n4 -fgcanalog.cfg* <Enter>

is the command you would type on the command line to handle 4 inbound analog network calls on the four analog channels defined in the configuration file shown in *Figure 10. Analog (gcanalog.cfg) Technology Configuration Sample File*.

For example: *outbound -n4 -fgcanalog.cfg* <Enter>

is the command you would type on the command line to handle 4 outbound analog network calls on the four analog channels defined in the configuration file shown in *Figure 10. Analog (gcanalog.cfg) Technology Configuration Sample File*.

7.2. Demo Programs for Windows NT

The following paragraphs describe multithreaded asynchronous (*gcmulti*) and synchronous (*gcmtsync_cui*) demonstration programs for handling inbound and outbound calls on a Windows NT platform.

The demonstration programs include complete source code in the installation directories. You may modify and rebuild a demo program using the Microsoft nmake utility or the Visual C++ version 4.x project workspace files. All the application files are included in the following directories:

- for the gcmulti asynchronous demo:
  - *\Program Files\Dialogic\Samples\gc_demos\gcmulti*
- for the gcmtsync_cui synchronous demo:
  - *\Program Files\Dialogic\Samples\gc_demos\gcmtsync_cui*

(NOTE: The *\Program Files\Dialogic* directory is the default directory. When installing Dialogic system software, a different directory can be specified.)

7.2.1. Multithreaded Asynchronous Demo Overview for Windows NT

The GlobalCall multithreaded asynchronous demonstration program (*gcmulti*) demonstrates handling inbound calls and outbound calls in asynchronous mode in a Windows NT environment. This demonstration program sets up all channels to either accept inbound calls or to make outbound calls.

When the accept-inbound calls mode is selected, the demo program looks at the last digit of the incoming DDI digits. When this last digit is an even number, the demo program simulates a “the time is” applet by playing a “the time is 9:30 a.m.” voice file and then disconnecting (hangs up). When this last digit is an odd number, the demo program runs a menu driven voice/facsimile application that, see *Figure 11. Multithreaded Asynchronous Demo, Call Processing*:

- plays an introductory voice file listing the menu selections and then
- gets the DDI digit entered in response to the voice menu.

The demo application responds to the DDI digit entered as described below and then disconnects:

Digit	Description
1	records the caller's message
2	plays the last message recorded
3	sends a fax if the demo platform includes a GammaLink fax product (currently unsupported)
4	receives and stores a fax if the demo platform includes a GammaLink fax product (currently unsupported)
5	plays a "good-bye" voice file

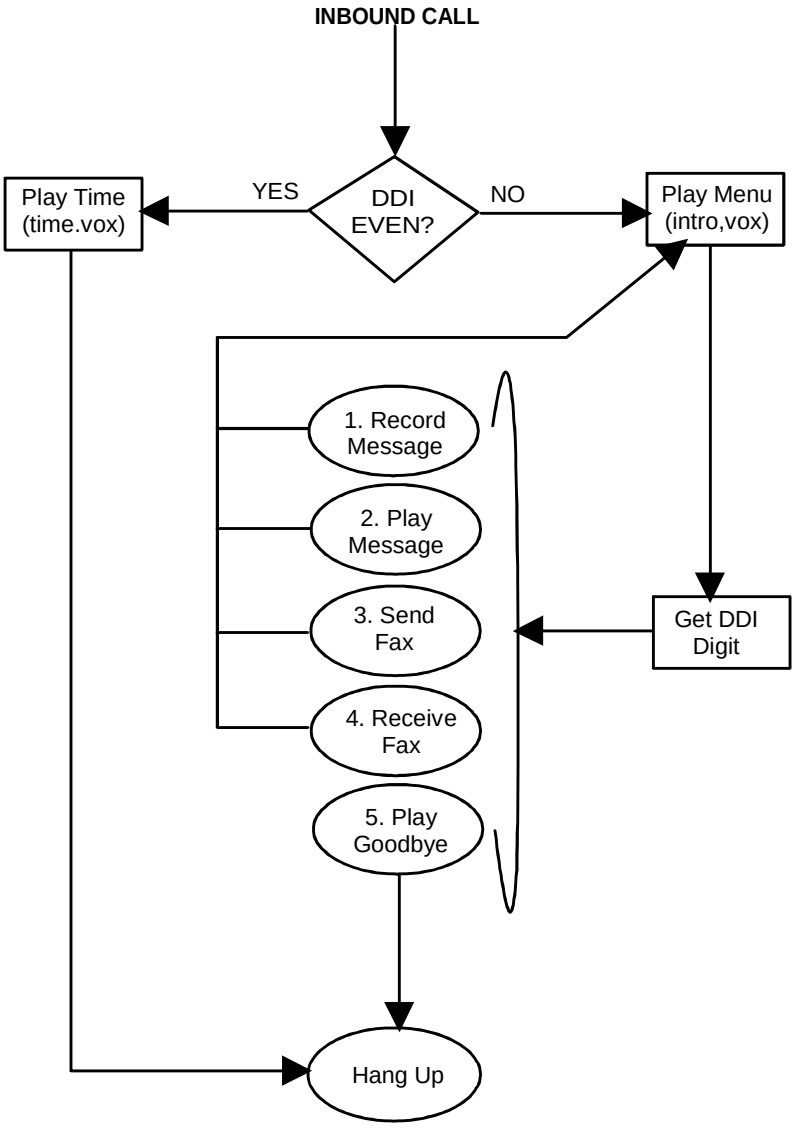

Figure 11. Multithreaded Asynchronous Demo, Call Processing

When the make-outbound calls mode is selected, all lines are used to place outbound calls. When the remote end answers, the demo program uses "the time is" applet to play a "the time is 9:30 a.m." voice file and then disconnects (hangs up) An outbound call is placed each time a channel changes to the IDLE state.

To run the demo program, see *paragraphs 7.2.3. Physical Connections for the Windows NT Demo, 7.2.4. Before Running the Windows NT Demo Programs*, and *7.2.5. Running the Asynchronous Windows NT Demo Program*.

7.2.2. Multithreaded Synchronous Demo Overview for Windows NT

The GlobalCall multithreaded synchronous demonstration program (*GCMtSync*) demonstrates handling inbound calls and outbound calls in synchronous mode in a Windows NT environment. The demonstration program implements a double layer state machine based on the GlobalCall API. The first layer deals with the GlobalCall call establishment and termination processes, see *Figure 12. Synchronous Demo, Call Establishment Process*. This layer includes the following call states:

Inbound Call States	Outbound Call States
GCST_NULL	GCST_NULL
GCST_IDLE	GCST_IDLE
GCST_OFFERED	GCST_DIALING
GCST_ACCEPTED	GCST_ALERTING
GCST_CONNECTED	GCST_CONNECTED
GCST_DISCONNECTED	GCST_DISCONNECTED

The second layer deals with the application states that can occur while the demo program handles the conversation portion of the call, see *Figure 13. Synchronous Demo, Application State Call Processing*, and includes the following call states:

Inbound Call States	Outbound Call States
APP_BLOCKED	APP_BLOCKED
APP_UNBLOCKED	APP_UNBLOCKED
APP_NULL	APP_NULL
APP_CONNECTED	APP_CONNECTED

Inbound Call States	Outbound Call States
APP_WELCOME	APP_DIALING
APP_RECORD	APP_PLAYBACK
APP_GETDIGIT	APP_DISCONNECTED
APP_PLAYBACK	APP_IDLE
APP_INVALID	
APP_GOODBYE	
APP_DISCONNECTED	
APP_IDLE	

Start the GlobalCall demo programs from the command line. Select the parameters and options you wish to use by typing the parameter value or option details after the appropriate option switch (see *Section 7.2.5. Running the Asynchronous Windows NT Demo* Program or *Section 7.2.6. Running the Synchronous Windows NT Demo Program*).

Figure 12. Synchronous Demo, Call Establishment Process illustrates the call states associated with handling inbound calls or setting up outbound calls in synchronous mode. All calls start from a GCST_NULL state.

For inbound calls and after receiving the GCEV_UNBOCKED event, the demo program issues a **gc_WaitCall( )** function in the GCST_NULL state to indicate readiness to accept an inbound call request. When the inbound call is received, the call state changes to GCST_OFFERED. In the GCST_OFFERED state, the call may be accepted by the demo program. From the GCST_OFFERED state, the call state changes to either the GCST_CONNECTED state or the GCST_ACCEPTED state. When the call is to be directly connected to a voice resource, a **gc_AnswerCall( )** function is issued to make the final connection. When the **gc_AnswerCall( )** function completes, the call changes to the GCST_CONNECTED state. If the demo program 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. When the **gc_AcceptCall( )** function completes, the call changes to the GCST_ACCEPTED state. To complete the connection, a **gc_AnswerCall( )** function is issued to make the final connection.

When the call completes, the demo program issues a **gc_DropCall( )** function that changes the call state to GCST_IDLE. A **gc_ReleaseCall( )** function is then issued to change the call state to GCST_NULL which establishes initial conditions for accepting the next inbound call or for making an outbound call. If a GCEV_DISCONNECTED event is received while the call is in the GCST_OFFERED, GCST_ACCEPTED or GCST_CONNECTED state, the demo program then issues a **gc_ReleaseCall( )** function, hangs up the ongoing call and then waits for the next call.

To make an outbound call and after receiving a GCEV_UNBLOCKED event, the demo program issues a **gc_MakeCall( )** function that requests that a call be made on a specific channel. The call enters the GCST_DIALING state and dialing information is sent to and acknowledged by the network. When the call is answered at the remote end, the **gc_MakeCall( )** function completes and the call changes to the GCST_CONNECTED state. If a GCEV_ALERTING event is received from the network indicating that the remote end has received the call but not yet answered the call, the call state changes to GCST_ALERTING. When the call is answered at the remote end, the **gc_MakeCall( )** function completes and the call changes to the GCST_CONNECTED state.

When the call disconnects, the demo program issues a **gc_DropCall( )** function that changes the call state to GCST_IDLE. A **gc_ReleaseCall( )** function is then issued to change the call state to GCST_NULL which establishes initial conditions for making the next outbound call or for accepting inbound calls. If a GCEV_DISCONNECTED event is received while the call is in the GCST_DIALING, GCST_ALERTING or GCST_CONNECTED state, the state changes to GCST_IDLE and the demo program then issues a **gc_ReleaseCall( )** function to return to the GCST_NULL state.

Figure 13. Synchronous Demo, Application State Call Processing illustrates demo application call states for processing inbound or outbound calls in the synchronous mode. If a GCEV_BLOCKED event is received during any application state, the application will halt its call processing activities and wait for a GCEV_UNBLOCKED event before continuing. When the demo application receives a GCEV_DISCONNECTED event while processing an inbound or an outbound call, the demo application issues a **gc_DropCall( )** function to change the call state to GCST_IDLE. The demo program then issues a **gc_ReleaseCall( )** function to return to the GCST_NULL state.

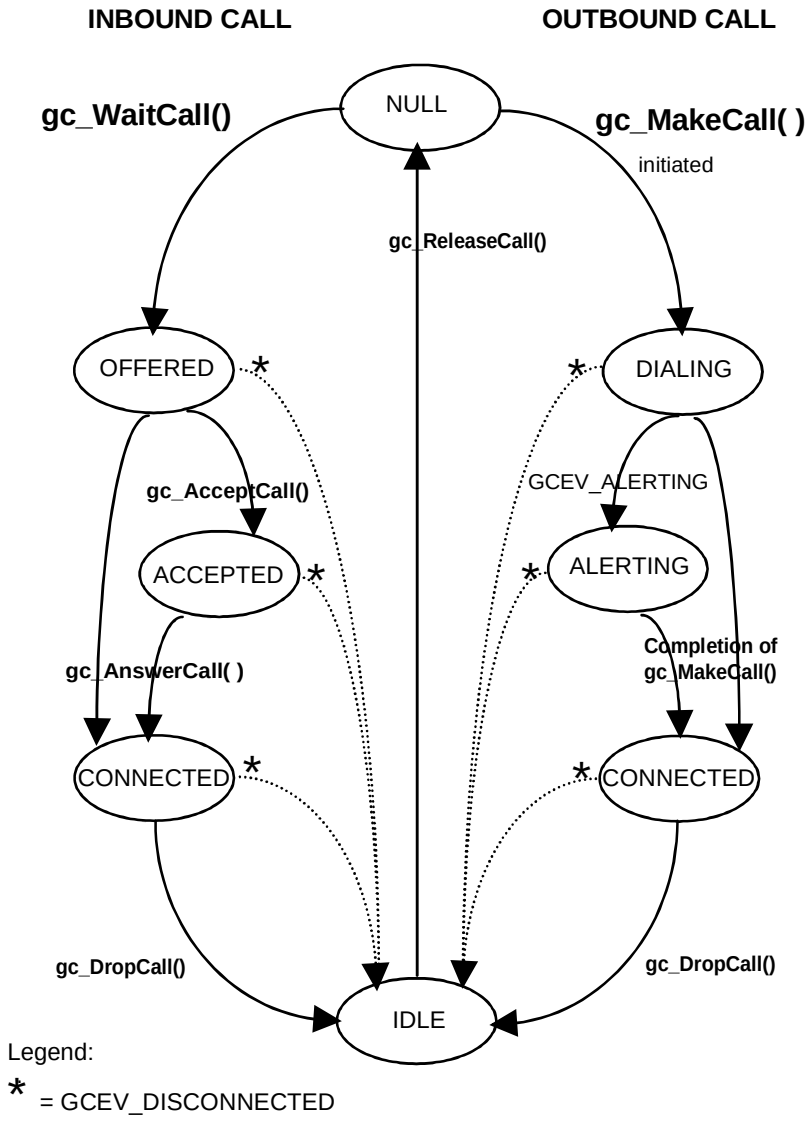

Figure 12. Synchronous Demo, Call Establishment Process

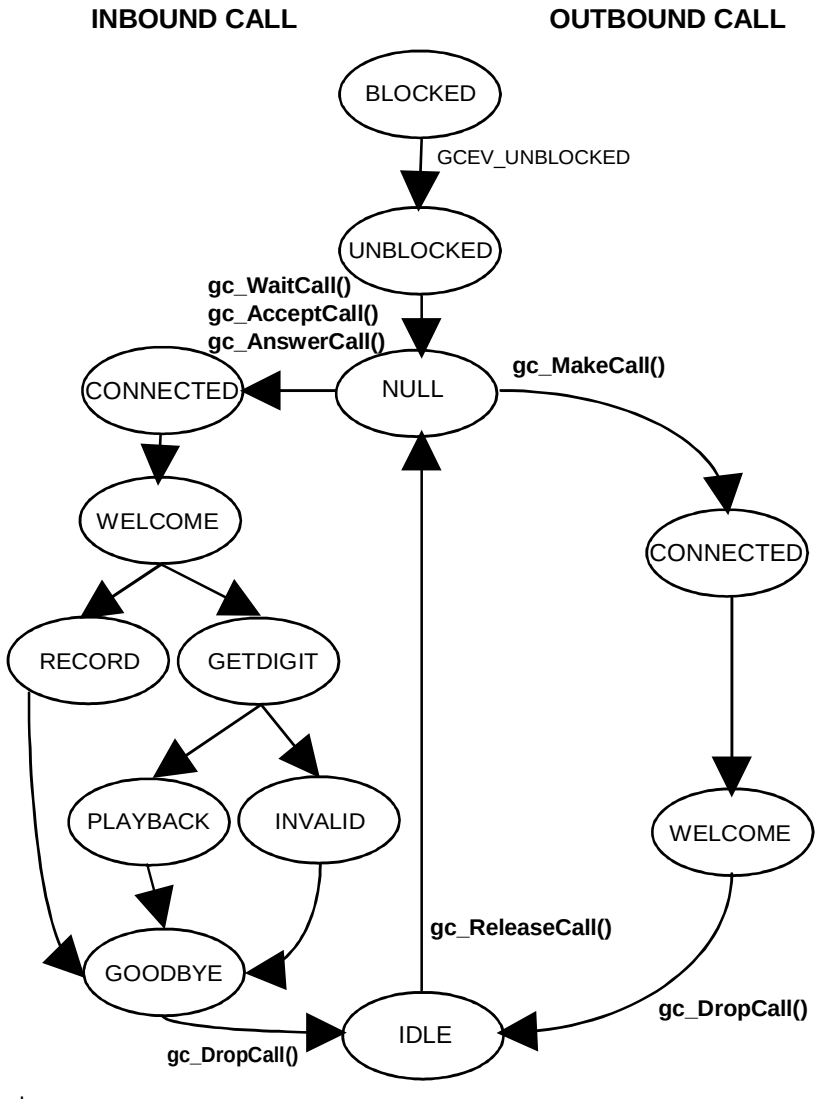

Figure 13. Synchronous Demo, Application State Call Processing

7.2.3. Physical Connections for the Windows NT Demo

To run the GlobalCall Demo programs, you need one of the following:

- a connection to the network (E-1 CAS/T-1 robbed bit or ISDN)
- an E-1, T-1 and/or ISDN simulator

You may make this connection either before or after installing the GlobalCall software.

7.2.4. Before Running the Windows NT Demo Programs

The demonstration programs include complete source code in the installation directories. You may modify and rebuild a demo program using the Microsoft nmake utility or the Visual C++ version 4.x project workspace files. All the application files are included in the following directories:

- for the gcmulti asynchronous demo:
  - *\Program Files\Dialogic\Samples\gc_demos\gcmulti*
- for the gcmtsync_cui synchronous demo:
  - *\Program Files\Dialogic\Samples\gc_demos\gcmtsync_cui*

(NOTE: The *\Program Files\Dialogic* directory is the default directory. When installing Dialogic system software, a different directory can be specified.)

The demo program can be compiled using either of the following methods:

- To use the Microsoft nmake utility, type:

  nmake -f <filename.mak>

- To use Visual C++ version 4.x, open a project workspace file <filename.mdp> from inside Visual C++ Integrated Development Environment and select:

  build/rebuild all

NOTE: Both ICAPI and ISDN call control libraries must be installed.

NOTE: A protocol must be installed on the system. Before running a demo program that uses a T-1 robbed bit protocol, disable the DTI Wait Call function in the *icapi.cfg* file. See the *icapi.cfg File paragraph* in the *GlobalCall E-1/T-1 Technology User's Guide* for details.

7.2.5. Running the Asynchronous Windows NT Demo Program

Start the asynchronous (*gcmulti*) demo program by typing the program name at the command line prompt, followed by the appropriate switch(es). The structure of the demo command is:

gcmulti -p<boardno><protocol> -n<#> -m<thread> -c<direction> [-f<disablefun>] -v<verbosity>

where:

-p<boardno><protocol>	boardno = board number (optional; default = 1) protocol = name of selected protocol (default = isdn)
-n<#>	# = number of connected lines/channels (default = 60, range is 1 to 60)
-m<thread>	m = multithreaded; s = single threaded (optional) (default = s)
-c<direction>	i = inbound call; o = outbound call (default = i)
-f<disablefun>	(optional) disable function, where *disablefun* values are: ANI to disable **gc_GetANI( )** CALLACK to disable **gc_CallAck( )** SETCALLNUM to disable **gc_SetCallingNum( )**
-v<verbosity>	1 = display error messages plus call setup and call tear down messages such as generated during a make call, drop call, answer call and accept call activity. (default = 1) 2 = 1 + display all initialization messages generated during application start up and all closing messages generated when the application exits.

3 = 2 + display all messages generated in conjunction with function calls and all events received including termination events and unexpected events.

-f<disablefun>, disables function calls that are coded into the demo program but are not supported by the protocol being run. See the *Limitations paragraph* in the *GlobalCall Country Dependent Parameters (CDP) Reference* for the protocol installed. Separate *-f<disablefun>* switches must be entered for each function to be disabled.All information displayed on the screen can be rerouted to a log file by appending a >filename parameter to the end of the *gcmulti* command line; (e.g., *gcmulti -p1isdn -n30 -ci -v2 >demo.log)* and then pressing the **Enter** key. Note that ALL information will be sent to the file specified and the display will remain blank until you press the **Esc** key to close the demo program.

For example: *gcmulti -p1isdn -n30 -ci -v2* <Enter>

is the command you would type on the command line to run the single-threaded asynchronous demonstration program on board 1 using ISDN inbound protocol and 30 channels with a verbosity level of 2.

Pressing the **Esc** key closes the demo program. When the demo program closes, call information for each channel, see *Figure 14. Demo Call Information Example*, total errors and the total calls handled are calculated and displayed. The calls per channel are displayed in the format:

- #)Calls[#]

where: #) = the channel number (e.g., 1, 2, etc.) and [#] = the total number of calls completed by that channel during the time the demo program ran.

The [SYS] Total errors = value is the total number of errors that occurred during the time the demo program ran.

The [SYS] Total calls = value is the total number of calls completed by all channels during the time the demo program ran. That is, the summation: # Calls on Channel 1 + # Calls on Channel 2 + … + # Calls on Channel n.

```
Call Information
1)Calls[10]   2)Calls[10]   3)Calls[10]   4)Calls[10]
5)Calls[10]   6)Calls[10]   7)Calls[10]   8)Calls[10]
9)Calls[10]  10)Calls[10]  11)Calls[10]  12)Calls[10]
13)Calls[10] 14)Calls[10]  15)Calls[10]  16)Calls[10]
17)Calls[10] 18)Calls[10]  19)Calls[10]  20)Calls[10]
21)Calls[10] 22)Calls[10]  23)Calls[10]  24)Calls[10]
25)Calls[10] 26)Calls[10]  27)Calls[10]  28)Calls[10]
29)Calls[10] 30)Calls[10]
[SYS] Total errors = 0
[SYS] Total calls = 300
```

Figure 14. Demo Call Information Example

7.2.6. Running the Synchronous Windows NT Demo Program

Start the synchronous (*gcmtsync_cui*) demo program by typing the program name at the command line prompt, followed by the appropriate switch(es). The structure of the demo command is:

gcmtsync_cui -n<numlines> -p<brdnum><protocol> -l<loglevel> -[i|o]

where:

-n<numlines> Number of connected lines/channels (default = 60, range is 1 to 60)

-p<brdnum><protocol> Selected protocol on selected board number “brdnum”, where “brdnum” must be set to 1 or 2 (for a single board, set to 1). Each board to be used by the demo program must be opened by including a separate -p<brdnum><protocol> switch to open that board.

-l<loglevel> specifies the logging level where “loglevel” is set to: 1 = logs high priority error messages, 2 = logs medium & high priority error messages or 3 = logs all error messages. If this parameter is not specified, the default is no logging

-[i/o]	i = inbound call; o = outbound call; a value must be specified.

For example: *gcmtsync_cui -n30 -p1br_r2_i -i -l3* <Enter>

is the command you would type on the command line to run the synchronous demonstration program on 30 channels using the inbound Brazil protocol for inbound calls with logging set to the highest logging level.

The following command would be entered on the command line to run the synchronous demonstration program on 60 channels using the inbound Brazil protocol with logging set to the highest logging level:

gcmtsync_cui -n60 -p1br_r2_i -p2br_r2_i -l3 -i

Appendix A
Summary of GlobalCall Functions and Events

Table 37. Summary of GlobalCall Functions

Function	Description
gc_AcceptCall( )	optional response to an incoming call request; used to indicate "ringing" to the remote end
gc_AnswerCall( )	response to an incoming call (equivalent to conventional "set hook off" function)
gc_Attach( )	logically connects a voice resource to a line device
gc_CallAck( )	enables user to control the response to an incoming call request by retrieving call information from the network. For ISDN PRI applications, **gc_CallAck( )** function is used in overlap receiving operation.
gc_CallProgress( )	notifies the network that the connection request is in progress.
gc_CCLibIDToName( )	converts call control library identification code to library name.
gc_CCLibNameToID( )	converts call control library name to library identification code
gc_CCLibStatus( )	retrieves status of the call control library specified
gc_CCLibStatusAll( )	retrieves status information for all call control libraries
gc_Close( )	closes a previously opened device and removes the channel from service

Function	Description
gc_CRN2LineDev( )	acquires the line device ID associated with a given CRN
gc_Detach( )	logically detaches a voice resource from the associated line device
gc_DropCall( )	disconnects a call; equivalent to a "hang-up"
gc_ErrorValue( )	returns the error value/failure reason related to the last GlobalCall function call. To process an error, this function must be called immediately after a GlobalCall function failed.
gc_GetANI( )	returns caller identification information
gc_GetBilling( )	gets the charge information for the call, after GCEV_DISCONNECTED event is received or **gc_DropCall( )** function is terminated
gc_GetCallInfo( )	gets information for the call
gc_GetCallState( )	acquires the state of the call associated with the CRN
gc_GetCRN( )	gets the CRN associated with a recently arrived event (such as GCEV_OFFERED)
gc_GetDNIS( )	gets the DNIS (DDI digits) associated with a specific CRN
gc_GetLineDev( )	gets the line device ID associated with a recently arrived event
gc_GetLinedevState( )	retrieves the status of the specified line device
gc_GetMetaEvent( )	transforms a call control library event (or any SRL event) into a GlobalCall metaevent

Function	Description
gc_GetMetaEventEx( )	(Windows NT extended asynchronous mode only) transforms a call control library event (or any SRL event) into a GlobalCall metaevent. Passes the SRL event handle to the application so that multithreaded applications can be implemented.
gc_GetNetworkH( )	returns network device handle associated with the specified line device
gc_GetParm( )	retrieves the parameter value specified for a line device
gc_GetUsrAttr( )	retrieves the attribute established using **gc_SetUsrAttr( )** function
gc_GetVer( )	returns the version number of the specified software component
gc_GetVoiceH( )	returns the voice device handle associated with the specified call control line device
gc_LoadDxParm( )	Sets voice parameters associated with a line device
gc_MakeCall( )	makes an outgoing call
gc_Open( )	opens a GlobalCall device and returns a unique line device handle to identify the physical device(s) that carry the call
gc_OpenEx( )	opens a GlobalCall device, sets a user defined attribute and returns a unique line device handle to identify the physical device(s) that carry the call. This function can be used in place of the **gc_Open( )** function followed by a **gc_SetUsrAttr( )** function.
gc_ReleaseCall( )	releases all internal resources for the specified call

Function	Description
gc_ReqANI( )	returns the caller's identification, normally included in the ISDN setup message and ANI-on-Demand requests
gc_ResetLineDev( )	disconnects any active calls on the line device; aborts all calls being setup
gc_ResultMsg( )	retrieves an ASCII string describing the result code
gc_ResultValue( )	returns the cause of an event
gc_SetBilling( )	for protocols that support this feature, sets billing information for the call
gc_SetCallingNum( )	sets the default calling party number on a specific line device; the calling party number thus defined will be used on all subsequent outbound calls
gc_SetChanState( )	sets a channel to the "in-service," "out-of-service," or "in-maintenance" state
gc_SetEvtMsk( )	sets the event mask associated with the specified line device
gc_SetInfoElem( )	enables setting an additional information element in the next outbound ISDN call
gc_SetParm( )	sets the default value of parameters used in call setup process
gc_SetUsrAttr( )	sets an attribute defined by the user
gc_SndMsg( )	sends non-call state-related ISDN message to network over the D channel while a call exists
gc_Start( )	starts all configured, call control libraries For UNIX applications, non-stub libraries are started.
gc_StartTrace( )	starts trace and places results in shared RAM
gc_Stop( )	stops all configured call control libraries started

Function	Description
gc_StopTrace( )	stops the trace and closes the file
gc_WaitCall( )	sets up conditions for processing incoming calls

Table 38. GlobalCall Event Summary

Event	Terminates	Ref	Description
GCEV_ACCEPT	**gc_AcceptCall( )**	CRN	Call received at remote end, but not yet answered
GCEV_ACKCALL	**gc_CallAck( )**	CRN	Indicates termination of **gc_CallAck( )** and that the DDI string may be retrieved by using **gc_GetDNIS( )**
GCEV_ALERTING	Unsolicited (enabled by default)	CRN	Destination party has answered call.
GCEV_ANSWERED	**gc_AnswerCall( )**	CRN	Call established and enters Connected state
GCEV_BLOCKED	Unsolicited (enabled by default)	LDID	Line is blocked and application cannot issue call-related function calls. Retrieve reason for line blockage using **gc_ResultValue( )**.
GCEV_CALLINFO	Unsolicited	CRN	Generated when an incoming information message is received.

Event	Terminates	Ref	Description
GCEV_CALLSTATUS	Unsolicited	CRN	Indicates that a timeout or a no answer (call control library dependent) condition was returned while the **gc_MakeCall( )** function is active
GCEV_CONGESTION	Unsolicited	CRN	Generated when an incoming congestion message is received.
GCEV_CONNECTED	**gc_MakeCall( )**	CRN	Call is connected
GCEV_D_CHAN_ STATUS	Unsolicited	LDID	Generated when the status of the D channel changes.
GCEV_DISCONNECTED	Unsolicited	CRN	Call disconnected by remote end.
	Any request or message rejected by network or that has timed-out	Either CRN or LDID	The error detected prevents further call processing on this call.
GCEV_DIVERTED	Unsolicited	CRN	Received request to call forward using DPNSS protocol.
GCEV_DROPCALL	**gc_DropCall( )**	CRN	Call is disconnected and call enters Idle state
GCEV_FACILITY	Unsolicited	LDID	Generated when an incoming facility message is received.

Event	Terminates	Ref	Description
GCEV_FACILITY_ACK	Unsolicited	LDID	Generated when an incoming facility ACK message is received.
GCEV_FACILITY_REJ	Unsolicited	LDID	Generated when an incoming facility reject message is received.
GCEV_HOLDACK	**gc_HoldCall( )**	CRN	Generated when an acknowledgement is sent in response to a hold call message.
GCEV_HOLDCALL	Unsolicited	CRN	Generated when a hold current call message is received.
GCEV_HOLDREJ	**gc_HoldCall( )**	CRN	Generated when a hold call request is rejected and the hold call reject message is sent to remote end.
GCEV_ISDNMSG	Unsolicited	CRN	Generated when an incoming unrecognized ISDN message is received.
GCEV_L2BFFRFULL	Unsolicited	CRN	Generated when the incoming layer

Event	Terminates	Ref	Description
			2 access message buffer is full. (reserved for future use)
GCEV_L2FRAME	Unsolicited	CRN	Generated when an incoming layer 2 access message is received.
GCEV_L2NOBFFR	Unsolicited	CRN	Generated when no free space is available for an incoming layer 2 access message.
GCEV_NOTIFY	Unsolicited	CRN	Generated when an incoming notify message is received.
GCEV_NSI	Unsolicited	CRN	Generated when a Network Specific Information (NSI) message is received using DPNSS protocol.
GCEV_OFFERED	Unsolicited	CRN	Inbound call arrived; call enters Offered state.
GCEV_PROCEEDING	Unsolicited (enabled by default)	CRN	Generated when an incoming proceeding message is received.
GCEV_PROGRESSING	Unsolicited (enabled by default)	CRN	Generated when an incoming progress message

Event	Terminates	Ref	Description
			is received.
GCEV_REQANI	**gc_ReqANI( )**	CRN	Generated when ANI information is received from network.
GCEV_RESETLINEDEV	**gc_ResetLineDev( )**	LDID	Disconnects any active calls on the line device.
GCEV_RETRIEVEACK	**gc_RetrieveCall( )**	CRN	Generated when an acknowledgement is sent in response to a retrieve hold call message.
GCEV_RETRIEVECALL	Unsolicited	CRN	Generated when a retrieve hold call message is received.
GCEV_RETRIEVEREJ	**gc_RetrieveCall( )**	CRN	Generated when a rejection message is sent in response to a request to retrieve held call.
GCEV_SETBILLING	**gc_SetBilling( )**	CRN	Generated when billing information for the call is acknowledged by the network.
GCEV_SETCHANSTATE	**gc_SetChanState( )** or unsolicited	CRN	Sets operating state of channel. Or if an unsolicited event, generated when the status of the

Event	Terminates	Ref	Description
			B channel changes or a maintenance message is received from the network.
GCEV_SETUP_ACK	Unsolicited (disabled by default)	CRN	Generated when an incoming setup ACK message is received.
GCEV_TASKFAIL	Unsolicited	Either CRN or LDID	An unsolicited error event occurred during the execution of a function.
GCEV_TRANSFERACK	Unsolicited	CRN	Generated when an acknowledgement is sent in response to a transfer call to another destination message using DPNSS protocol.
GCEV_TRANSFERCALL	Unsolicited	CRN	Generated when a transfer call to another destination message is received.
GCEV_TRANSFERREJ	Unsolicited	CRN	Generated when a rejection message is sent in response to a request to transfer call to another destination using DPNSS protocol.

Event	Terminates	Ref	Description
GCEV_TRANSIT	Unsolicited	CRN	Generated when a message is sent via a call transferring party to the destination party after a transfer call connection is completed using DPNSS protocol.
GCEV_UNBLOCKED	Unsolicited (enabled by default)	LDID	Line is unblocked. Application may issue call-related commands to this line device.
GCEV_USRINFO	Unsolicited	CRN	Generated when an incoming User-to-User Information (UUI) message is received.

Appendix B
GlobalCall Error Code & Result Value Summary

Table 39. GlobalCall Error Code Summary

Error Code Returned	Description
EGC_ALARM	Function interrupted by alarm
EGC_ALARMDBINIT	Alarm database failed to initialize
EGC_ATTACHED	Specified resource already attached
EGC_BADFCN	Wrong function code (TSR)
EGC_BUSY	Line is busy
EGC_CCLIBSPECIFIC	Error specific to call control library
EGC_CCLIBSTART	At least one call control library failed to start
EGC_CEPT	Operator intercept detected
EGC_COMPATIBILITY	Incompatible components
EGC_CPERROR	SIT detection error
EGC_DEVICE	Invalid device handle
EGC_DIALTONE	No dial tone detected
EGC_DRIVER	Driver error
EGC_DTOPEN	**dt_open( )** function failed
EGC_DUPENTRY	Duplicate entry inserted into GlobalCall database
EGC_FILEOPEN	Error opening file
EGC_FILEREAD	Error reading file

Error Code Returned	Description
EGC_FILEWRITE	Error writing file
EGC_FUNC_NOT_ DEFINED	Protocol function not defined
EGC_GC_STARTED	GlobalCall library is already started
EGC_GCDBERR	GlobalCall database error
EGC_GCNOTSTARTED	GlobalCall not started
EGC_ILLSTATE	Function is not supported in the current state
EGC_INTERR	Internal GlobalCall Error
EGC_INVCRN	Invalid call reference number
EGC_INVDEVNAME	Invalid device name
EGC_INVLINEDEV	Invalid line device passed
EGC_INVMETAEVENT	Invalid metaevent
EGC_INVPARM	Invalid parameter (argument)
EGC_INVPROTOCOL	Invalid protocol name
EGC_INVSTATE	Invalid state
EGC_LINERELATED	Error is related to line device
EGC_MAXDEVICES	Exceeded maximum devices limit
EGC_NAMENOTFOUND	Trunk device name not found
EGC_NDEVICE	Too many devices opened
EGC_NOANSWER	Rang called party, called party did not answer
EGC_NOCALL	No call was made or transferred
EGC_NOERR	No error
EGC_NOMEM	Out of memory

Error Code Returned	Description
EGC_NORB	No ringback detected
EGC_NOT_INSERVICE	Called number is not in-service
EGC_NOVOICE	Call needs voice resource, use **gc_Attach( )** function
EGC_NPROTOCOL	Too many protocols opened
EGC_OPENH	Error opening voice channel
EGC_PFILE	Error opening parameter file
EGC_PROTOCOL	Protocol error
EGC_PUTEVT	Error queuing event
EGC_SETALRM	Set alarm mode failed
EGC_SRL	SRL failure
EGC_STOPD	Call progress stopped
EGC_SYNC	Set mode flag to EV_ASYNC instead of EV_SYNC
EGC_SYSTEM	System error
EGC_TASKABORTED	Task aborted
EGC_TIMEOUT	Function time out
EGC_TIMER	Error starting timer
EGC_TSRNOTACTIVE	cclib not active (TSR)
EGC_UNSUPPORTED	Function is not supported by this technology
EGC_USER	Function interrupted by user
EGC_USRATTRNOTSET	User attribute for this line device was not set
EGC_VOICE	No voice resource attached

Error Code Returned	Description
EGC_VOXERR	Error from voice software
EGC_XMITALRM	Send alarm failed

Table 40. GlobalCall Result Value Summary

Result Value	Description
GCRV_ALARM	Event caused by alarm
GCRV_B8ZSD	Bipolar eight zero substitution detected
GCRV_B8ZSDOK	Bipolar eight zero substitution detected recovered
GCRV_BPVS	Bipolar violation count saturation
GCRV_BPVSOK	Bipolar violation count saturation recovered
GCRV_BUSY	Line is busy
GCRV_CCLIBSPECIFIC	Event caused by specific call control library failure
GCRV_CECS	CRC4 error count saturation
GCRV_CECSOK	CRC4 error count saturation recovered
GCRV_CEPT	Operator intercept detected
GCRV_CPERROR	SIT detection error
GCRV_DIALTONE	No dial tone detected
GCRV_DPM	Driver performance monitor failure
GCRV_DPMOK	Driver performance monitor failure recovered
GCRV_ECS	Error count saturation
GCRV_ECSOK	Error count saturation recovered

Result Value	Description
GCRV_FERR	Frame bit error
GCRV_FERROK	Frame bit error recovered
GCRV_FSERR	Frame sync error
GCRV_FSERROK	Frame sync error recovered
GCRV_INTERNAL	Event caused internal failure
GCRV_LOS	Initial loss of signal detection
GCRV_LOSOK	Initial loss of signal detection recovered
GCRV_MFSERR	Received multi frame sync error
GCRV_MFSERROK	Received multi frame sync error recovered
GCRV_NOANSWER	Event caused by no answer
GCRV_NORB	No ringback detected
GCRV_NORMAL	Normal completion
GCRV_NOT_INSERVICE	Called number is not in-service
GCRV_NOVOICE	Call needs voice resource, use **gc_Attach( )** function
GCRV_OOF	Out of frame error, count saturation
GCRV_OOFOK	Out of frame error, count saturation recovered
GCRV_PROTOCOL	Event caused by protocol error
GCRV_RBL	Received blue alarm
GCRV_RBLOK	Received blue alarm recovered
GCRV_RCL	Received carrier loss
GCRV_RCLOK	Received carrier loss recovered
GCRV_RDMA	Received distant multi-frame alarm

Result Value	Description
GCRV_RDMAOK	Received distant multi-frame alarm recovered
GCRV_RED	Got a red alarm condition
GCRV_REDOK	Got a red alarm condition recovered
GCRV_RLOS	Received loss of sync
GCRV_RLOSOK	Received loss of sync recovered
GCRV_RRA	Remote alarm
GCRV_RRAOK	Remote alarm recovered
GCRV_RSA1	Received signaling all 1s
GCRV_RSA1OK	Received signaling all 1s recovered
GCRV_RUA1	Received unframed all 1s
GCRV_RUA1OK	Received unframed all 1s recovered
GCRV_RYEL	Received yellow alarm
GCRV_RYELOK	Received yellow alarm recovered
GCRV_SIGNALLING	Signaling change
GCRV_STOPD	Call progress stopped
GCRV_TIMEOUT	Event caused by time-out

Appendix C

GlobalCall Header Files

The GlobalCall header files, *gclib.h* and *gcerr.h* , listed in this appendix are for both Windows NT and UNIX. These header files apply to all technologies.

gclib.h Header File

```
/***************************************************************************
*
*    C Header:      gclib.h
*    Instance:      dnj25
*    Description:   GlobalCall header file for application use
*    %created_by:   wienerc %
*    %date_created: Tue Feb  10 15:59:54 1998 %
*
***************************************************************************/
/*************************************************************************
 *  Copyright (C) 1996-1998 Dialogic Corp.
 *  All Rights Reserved
 *
 *  THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF Dialogic Corp.
 *  The copyright notice above does not evidence any actual or
 *  intended publication of such source code.
 *************************************************************************/

#ifndef GCLIB_H
#define GCLIB_H

#ifndef lint
static char    *_dnj25_gclib_h = "@(#) %filespec: gclib.h-22.1.3 %  (%full_filespec:
gclib.h-22.1.3:incl:dnj25 %)";
#endif    /* !lint */

#ifndef DOS
#include <stdio.h>
#endif    /* !DOS */

#ifdef __cplusplus
extern "C" {   // C++ func bindings to enable funcs to be called from C++
#endif    /* __cplusplus */

#ifdef _WIN32
#pragma pack(1)
#endif    /* _WIN32 */

/*
 *                                 --- Rel Type: 0=Prod, 1=Beta, 2=Alpha, 3=Exp
 *                                | ----- Major Number
 *                                ||  ----- Minor Number
 *                                || |  ----- Beta Number
 *                                || | |  ----- Alpha Number
 *                                || | | |
 *                                vv v v v
 */
```

```
#ifdef unix
#define GC_VERSION   (long int)0x11030100
#else /* !unix */
#if defined(_WIN32) && defined(_M_IX86)
#define GC_VERSION   (long int)0x01000000
#else /* !_WIN32 && !_M_IX86 */
#if defined(_WIN32) && defined(_M_ALPHA)
#define GC_VERSION   (long int)0x21000001
#endif   /* _WIN32 && _M_ALPHA */
#endif   /* _WIN32 && _M_IX86 */
#endif   /* unix */



#define METAEVENT_MAGICNO 0xBAD012FBL
/*
 * This file will be exported to application
 */

/*
 * Typedefs used throughout GlobalCall software, and application.
 */
#define GCLIB_DEBUG_FILE_NAME   "gclib.dbg"     /* controls where debug file
                                                   list is kept */
#define MAX_BOARD_NAME_LENGTH   100             /* Not including the trailing
                                                   NULL */
#define MAX_CCLIB_NAME_LENGTH   10              /* Not including the trailing
                                                   NULL */
#define LIBID_GC                 0              /* GlobalCall lib's id */
#define GC_MAX_CRNS_PER_LINEDEV 20

/*gc1*/
#define LINEDEV                         long
/*gc2*/
#define CRN                             long

/*
-- bit mask for gc_GetCCLibInfo
*/
#define GC_CCLIB_AVL                    0x1
#define GC_CCLIB_CONFIGURED             0x2
#define GC_CCLIB_FAILED                 0x4
#define GC_CCLIB_STUB                   0x8

/*
-- defines for gc_GetCallInfo()
*/
/*#define CALLED_SUBS                   0x5*/   /* In ISDN header files */
/*#define U_IES                         0x10*/  /* In ISDN header files */
#define CATEGORY_DIGIT                  0x100   /* Get category digit */
#define CONNECT_TYPE                    0x101   /* get callp connect type */
#define CALLNAME                        0x102   /* get caller name call ID */
#define CALLTIME                        0x103   /* get caller time call ID */

/*
-- Defines for the connect types
*/
#define GCCT_NA   0             /* call progress is not available */
#define GCCT_CAD  1             /* connect due to cadence */
#define GCCT_LPC  2             /* connect due to loop current */
#define GCCT_PVD  3             /* connect due to positive voice detection */
```

```
#define GCCT_PAMD 4              /* connect due to positive answering machine
                                    detection */
#define GCCT_FAX   5                /* connect due to FAX */


/*gc3*/
/*
-- Note: this structure is intended to be used in the future
-- by gc_Start(), but is not yet implemented
*/
typedef struct {
     int       rfu;
} GC_START_STRUCT, *GC_START_STRUCTP;

typedef struct {
   int      num_avllibraries;
   int      num_configuredlibraries;
   int      num_failedlibraries;
   int      num_stublibraries;

   /*
   -- these are an array of strings, each string terminated with a NULL
   -- e.g. avl_libraries[0] = "ICAPI"
   --      avl_libraries[1] = "ISDN"
   --      avl_libraries[2] = "ANAPI"
   */
   char     **avllibraries;
   char     **configuredlibraries;
   char     **failedlibraries;
   char     **stublibraries;
} GC_CCLIB_STATUS, *GC_CCLIB_STATUSP;


typedef struct {
   long     poll_units;               /* # of poll units before gc_GetEvent */
                                      /* should return */
                                      /* -1 = no limit */
                                      /* 0 is the same as 1 */
                                      /* NB - Only 1 or forever is currently */
                                      /* implemented */
                                      /* in the future, poll_units may be */
                                      /* either cycles or time */
   int      rfu1;                     /* reserved for future use */
   int      rfu2;                     /* reserved for future use */
} GETEVENT;

/*
-- for gc_Open() successful "termination event"
-- need to distinguish from normal BLOCKED/UNBLOCKED codes
*/
#define     GCEV_OPEN_UNBLOCKED     -1L /* result of open being successful */
                                        /* a - number so not to conflict with */
                                        /* DTILIB codes */


/*
 * Data structure types
 */
#define GCME_UNKNOWN_STRUCT_TYPE        0
#define GCME_EVTBLK_STRUCT_TYPE         1
#define GCME_EVTDATA_STRUCT_TYPE        2
```

```
/*
 * Defines for GlobalCall API event codes
 */
#ifndef DOS
#define DT_GC        0x800
#else     /* DOS */
#define DT_GC        0x2000
#endif    /* !DOS */

#ifdef DOS
#define DV_GCAPI (DT_GC | 100)           /* DV_ICAPI is 100 */
#endif    /* DOS */

/*gc4*/
#define GCEV_TASKFAIL      (DT_GC | 0x01) /* Abnormal condition; state unchanged
                                           */
#define GCEV_ANSWERED      (DT_GC | 0x02) /* Call answered and connected */
#define GCEV_CALLPROGRESS  (DT_GC | 0x03)
#define GCEV_ACCEPT        (DT_GC | 0x04) /* Call is accepted */
#define GCEV_DROPCALL      (DT_GC | 0x05) /* gc_DropCall is completed */
#define GCEV_RESETLINEDEV  (DT_GC | 0x06) /* Restart event */
#define GCEV_CALLINFO      (DT_GC | 0x07) /* Info message received */
#define GCEV_REQANI        (DT_GC | 0x08) /* gc_ReqANI() is completed */
#define GCEV_SETCHANSTATE  (DT_GC | 0x09) /* gc_SetChanState() is completed */
#define GCEV_FACILITY_ACK  (DT_GC | 0x0A)
#define GCEV_FACILITY_REJ  (DT_GC | 0x0B)
#define GCEV_MOREDIGITS    (DT_GC | 0x0C) /* cc_moredigits() is completed*/
#define GCEV_SETBILLING    (DT_GC | 0x0E) /* gc_SetBilling() is completed */
#define GCEV_ALERTING      (DT_GC | 0x21) /* The destination telephone terminal
                                           * equipment has received connection
                                           * request (in ISDN accepted the
                                           * connection request.  This event is
                                           * an unsolicited event
                                           */
#define GCEV_CONNECTED     (DT_GC | 0x22) /* Destination answered the request */
#define GCEV_ERROR         (DT_GC | 0x23) /* unexpected error event */
#define GCEV_OFFERED       (DT_GC | 0x24) /* A connection request has been made
                                           */
#define GCEV_DISCONNECTED  (DT_GC | 0x26) /* Remote end disconnected */
#define GCEV_PROCEEDING    (DT_GC | 0x27) /* The call state has been changed to
                                           * the proceeding state */
#define GCEV_PROGRESSING   (DT_GC | 0x28) /* A call progress message has been
                                           * received */
#define GCEV_USRINFO       (DT_GC | 0x29) /* A user to user information event is
                                           * coming */
#define GCEV_FACILITYREQ   (DT_GC | 0x2A) /* A facility request is made by CO */
                                          /* NB: ISDN equivalent value is */
                                          /* CCEV_FACILITY */
#define GCEV_CONGESTION    (DT_GC | 0x2B) /* Remote end is not ready to accept
                                           * incoming user information */
#define GCEV_FACILITY      (DT_GC | 0x2C) /* Facility info. available */
#define GCEV_D_CHAN_STATUS (DT_GC | 0x2E) /* Report D-channel status to the user */
#define GCEV_NOUSRINFOBUF  (DT_GC | 0x30) /* User information element buffer is
                                           * not ready */
#define GCEV_NOFACILITYBUF (DT_GC | 0x31) /* Facility buffer is not ready */
#define GCEV_BLOCKED       (DT_GC | 0x32) /* Line device is blocked */
#define GCEV_UNBLOCKED     (DT_GC | 0x33) /* Line device is no longer blocked */
#define GCEV_ISDNMSG       (DT_GC | 0x34)
#define GCEV_NOTIFY        (DT_GC | 0x35) /* Notify message received */
#define GCEV_L2FRAME       (DT_GC | 0x36)
#define GCEV_L2BFFRFULL    (DT_GC | 0x37)
```

```
#define GCEV_L2NOBFFR      (DT_GC | 0x38)
#define GCEV_SETUP_ACK     (DT_GC | 0x39)
#define GCEV_CALLSTATUS    (DT_GC | 0x3A) /* call status, e.g. busy */

#ifdef _WIN32
/*gc5*/
/* these events only apply to those sites using ISDN DPNSS */
#define GCEV_DIVERTED      (DT_GC | 0x40)
#define GCEV_HOLDACK       (DT_GC | 0x41)
#define GCEV_HOLDCALL      (DT_GC | 0x42)
#define GCEV_HOLDREJ       (DT_GC | 0x43)
#define GCEV_RETRIEVEACK   (DT_GC | 0x44)
#define GCEV_RETRIEVECALL  (DT_GC | 0x45)
#define GCEV_RETRIEVEREJ   (DT_GC | 0x46)
#define GCEV_NSI           (DT_GC | 0x47)
#define GCEV_TRANSFERACK   (DT_GC | 0x48)
#define GCEV_TRANSFERREJ   (DT_GC | 0x49)
#define GCEV_TRANSIT       (DT_GC | 0x4A)

/* end of ISDN DPNSS specific */
#endif    /* _WIN32 */
#define GCEV_ACKCALL       (DT_GC | 0x50) /* Termination event for gc_CallACK()
                                           */


/*
 * MASK defines which may be modified by gc_SetEvtMsk().
 * These masks are used to mask or unmask their corresponding events,
 * GCEV_xxxx.
 */
#define GCMSK_ALERTING      0x01
#define GCMSK_PROCEEDING    0x02
#define GCMSK_PROGRESS      0x04
#define GCMSK_NOFACILITYBUF 0x08
#define GCMSK_NOUERINFO     0x10
#define GCMSK_BLOCKED       0x20
#define GCMSK_UNBLOCKED     0x40
#define GCMSK_PROC_SEND     0x80
#define GCMSK_SETUP_ACK     0x100



/*
 * Event Mask Action values
 *
 */
#define GCACT_SETMSK        0x01          /* Enable notification of events
                                           * specified in bitmask and disable
                                           * notification of previously set
                                           * events */
#define GCACT_ADDMSK        0x02          /* Enable notification of events
                                           * specified in bitmask in addition
                                           * to previously set events. */
#define GCACT_SUBMSK        0x03          /* Disable notification of events
                                           * specified  in bitmask. */

/*
 * BUFFER sizes
 */
#define GC_BILLSIZE         0x60          /* For storing billing info */
#define GC_ADDRSIZE         0x30          /* For storing ANI or DNIS digits. */
```

```
/*
 * Components supported for gc_GetVer()
 */
#define GCGV_LIB            0             /* GlobalCall library */
#define ICGV_LIB            1             /* ICAPI library */
#define ISGV_LIB            2             /* ISDN library */
#define ANGV_LIB            3             /* ANAPI library */


/*
 * Cause definitions for dropping a call
 */
#define GC_UNASSIGNED_NUMBER       0x01   /* Number unassigned / unallocated */
#define GC_NORMAL_CLEARING         0x10   /* Call dropped under normal conditions*/
#define GC_CHANNEL_UNACCEPTABLE    0x06
#define GC_USER_BUSY               0x11   /* End user is busy */
#define GC_CALL_REJECTED           0x15   /* Call was rejected */
#define GC_DEST_OUT_OF_ORDER       0x19   /* Destination is out of order */
#define GC_NETWORK_CONGESTION      0x2a
#define GC_REQ_CHANNEL_NOT_AVAIL   0x2c   /* Requested channel is not available
                                           */
#define GC_SEND_SIT                0x300  /* send Special Info. Tone (SIT) */

/*
 * RATE types for gc_SetBilling()
 */
#define GCR_CHARGE                 0x0000 /* Charge call (default) */
#define GCR_NOCHARGE               0x0100 /* Do not charge call */


/*
 * Defines for 'parm' parameter of gc_SetParm() and gc_GetParm()
 * gc6
 */
#define GCPR_ALARM                  1     /* Enable or disable alarm handling */
#define GCPR_WAITIDLE               2     /* Change wait for idle time-out */
#define GCPR_LOADTONES              4     /* Enable or disable loading tone */
#define GCPR_RINGBACKID             5     /* GTD id for ring back tone */
#define GCPR_OUTGUARD               6     /* maximum time for call progress */
#define GCPR_MINDIGITS              7     /* min # of digits */
#define GCPR_CALLINGPARTY      0x4001     /* set or get terminal phone number */
#define GCPR_CATEGORY            0x104    /* request caller category          */

#define GCPV_ENABLE                 1     /* enable feature */
#define GCPV_DISABLE                0     /* disable feature */

/*
 * Call States
 */
#define GCST_NULL                   0x00
#define GCST_ACCEPTED               0x01
#define GCST_ALERTING               0x02
#define GCST_CONNECTED              0x04
#define GCST_OFFERED                0x08
#define GCST_DIALING                0x10
#define GCST_IDLE                   0x20
#define GCST_DISCONNECTED           0x40


/*
```

```
 * Channel states which may be set using gc_SetChanState()
 */
#define GCLS_INSERVICE              0     /* Set channel to in service */
#define GCLS_MAINTENANCE            1     /* Set channel to maintenance state */
#define GCLS_OUT_OF_SERVICE         2     /* Set channel to out of service */


/*
 * Defines for gc_CallACK() when getting more digits.
 */

#define GCIF_DDI                1       /* get additional DDI digits */


/*gc7*/
#define GCDG_COMPLETE           0x0000   /* No more digits after that */
#define GCDG_PARTIAL            0x0100   /* Maybe more digits after that */

#define GCDG_NDIGIT             0x00FF   /* Get infinite string of digits */
#define GCDG_MAXDIGIT           0x00E    /* maximum # of DDI digits which can
                                            be collected */

/*
-- Defines for gc_GetLinedevState
*/
#define GCGLS_BCHANNEL          0x0      /* B channel (ISDN) */
#define GCGLS_DCHANNEL          0x1      /* D channel (ISDN) */

#define GC_MAXNFACNETWORKID 251          /* Maximum non-facility network ID */

typedef struct {
/*
-- Note: structure is ordered with longest fields 1st
-- to improve access time with some compilers
*/
   long          magicno;               /* for internal validity check */

   /* application calls gc_GetMetaEvent() to fill in these fields */
   unsigned long  flags;                /* flags field */
                                        /* - possibly event data structure type
                                         */
                                        /* i.e. evtdata_struct_type */
   void          *evtdatap;             /* pointer to the event data block */
                                        /* DOS will be of type EVTBLK for
                                           ICAPI,ISDN */
                                        /* other libraries to be determined */
                                        /* UNIX will be sr_getevtdatap() */
   long          evtlen;                /* event length */
                                        /* DOS - initially sizeof(EVTBLK) */
                                        /* UNIX sr_getevtlen */
#ifdef DOS
   long          devtype;               /* Specifies the product generating event */
   long          evtcode;               /* Event Code identifying the event */
   long          evtdata;               /* Data relevant to the event */
   long          devchan;               /* Device Channel: Channel Number on which
event occoured */
   long          board;                 /* Board Number: For non voice events
specifies board Number */
#else    /* !DOS */
   long          evtdev;                /* UNIX - sr_getevtdev */
   long          evttype;               /* Event type */
#endif   /* DOS */
```

```
   /* linedev & crn are only valid for GlobalCall events */
   LINEDEV        linedev;               /* linedevice */
   CRN            crn;                   /* crn - if 0 then no crn for this event */
   long           rfu2;                  /* for future use only */

   void           *usrattr;              /* user attribute */
   int            cclibid;               /* ID of CCLib that event is associated
                                            with */
                                         /* + = CCLib ID number */
                                         /* -1 = unknown */
#ifdef DOS
   int            evt_data_struct_type;  /* event data structure type */
#else    /* !DOS */
   int            rfu1;                  /* for future use only */
#endif   /* DOS */

} METAEVENT, *METAEVENTP;

/* define(s) for flags field within METAEVENT structure */
#define GCME_GC_EVENT      0x1            /* Event is a GlobalCall event */

/*gc8*/
#define MAXPHONENUM        32


/* this structure is for future use */
typedef struct  {
   long                    flags;
   long                    connecttype;
} GCLIB_MAKECALL_BLK;

typedef struct {
   GCLIB_MAKECALL_BLK      *gclib;        /* GlobalCall specific portion */
   void                    *cclib;        /* cclib specific portion */
} GC_MAKECALL_BLK,  *GC_MAKECALL_BLKP;

typedef union {
  struct {
    long cents;
  } ATT, *ATT_PTR;
} GC_RATE_U,  *GC_RATE_U_PTR;


typedef struct {
   long                    flags;
   long                    rfu;
} GCLIB_WAITCALL_BLK;

typedef struct {
   GCLIB_WAITCALL_BLK      *gclib;        /* GlobalCall specific portion */
   void                    *cclib;
/* cclib specific portion */
} GC_WAITCALL_BLK,  *GC_WAITCALL_BLKP;


/* define(s) for type field within GC_CALLACK_BLK structure */
#define  GCACK_SERVICE_DNIS   0x1
#define  GCACK_SERVICE_ISDN   0x2

typedef struct {
   unsigned long type;   /* type of a structure inside following union */
   long rfu;             /* will be used for common functionality */
```

```
   union {
      struct {
    int accept;
      } dnis;
      struct {
         int acceptance;
         /* 0x0000 proceding with the same B chan */
         /* 0x0001 proceding with the new B chan */
         /* 0x0002 setup ACK */
         LINEDEV linedev;
      } isdn;
      struct {
    long gc_private[4];
      } gc_private;
   } service;           /* what kind of service is requested */
         /* related to type field */
} GC_CALLACK_BLK,  *GC_CALLACK_BLK_PTR;


typedef union {
  short         shortvalue;
  long          longvalue;
  int           intvalue;
  char          charvalue;
  char          *paddress;
  void          *pstruct;
} GC_PARM;

/* structure for gc_GetDeviceNameInfo */
typedef struct {
   int          cclibid;
   long         rfu;
} GC_DEVICENAME_INFO, *GC_DEVICENAME_INFOP;



/*
-- structures for gc_SndMsg
-- This structure is an rfu
*/
typedef struct {
   long         flags;
   long         rfu;
} GCLIB_IE_BLK, *GCLIB_IE_BLKP;

typedef struct {
   GCLIB_IE_BLK *gclib;                  /* GlobalCall specific portion */
   void         *cclib;                  /* cclib specific portion */
} GC_IE_BLK, *GC_IE_BLKP;


/*
-- structures for gc_SndFrame
-- This structure is an rfu
*/
typedef struct {
   long         flags;
   long         rfu;
} GCLIB_L2_BLK, *GCLIB_L2_BLKP;

typedef struct {
```

```
   GCLIB_L2_BLK *gclib;                  /* GlobalCall specific portion */
   void         *cclib;                  /* cclib specific portion */
} GC_L2_BLK, *GC_L2_BLKP;


/*
 * GlobalCall Function Prototypes
 * Note: New functions will need to be addeded twice: once for
 * _MSC_VER ... & once for not
 */
#if ( defined (_MSC_VER) || defined (__STDC__) || defined (__cplusplus) )

#if defined (__cplusplus)
extern "C" {
#endif    /* __cplusplus */
int gc_AcceptCall(CRN crn, int rings, unsigned long mode);
int gc_AnswerCall(CRN crn, int rings, unsigned long mode);
int gc_Attach(LINEDEV linedev, int voiceh, unsigned long mode);
int gc_CallAck(CRN crn, GC_CALLACK_BLK *callack_blkp, unsigned long mode);
int gc_CallProgress(CRN crn, int indicator);
int gc_CCLibIDToName(int cclibid, char **lib_name);
int gc_CCLibNameToID(char *lib_name, int *cclibidp);
int gc_CCLibStatus(char *cclib_name, int *cclib_infop);
int gc_CCLibStatusAll(GC_CCLIB_STATUS *cclib_status);
int gc_Close(LINEDEV linedev);
int gc_CRN2LineDev(CRN crn, LINEDEV *linedevp);
int gc_Detach(LINEDEV linedev, int voiceh, unsigned long mode);
int gc_DropCall(CRN crn, int cause, unsigned long mode);
int gc_ErrorValue(int *gc_errorp, int *cclibidp, long *cclib_errorp);
int gc_ExtensionFunction(int cclibid, LINEDEV linedev, CRN crn, void *datap);
int gc_GetANI(CRN crn, char * anibuf);
int gc_GetBilling(CRN crn, char *billing_buf);
int gc_GetCallInfo(CRN crn,int info_id, char *valuep);
int gc_GetCallState(CRN crn, int *state_ptr);
int gc_GetCRN(CRN *crn_ptr, METAEVENT *metaeventp);
int gc_GetDeviceNameInfo(char *DeviceName, GC_DEVICENAME_INFOP devicename_infop);
int gc_GetDlgerrValue( LINEDEV linedev, int *dlgerrp);
int gc_GetDNIS(CRN crn, char *dnis);
#ifdef DOS
int gc_GetEvent(GETEVENT *geteventp, METAEVENT *metaeventp);
#endif    /* DOS */
#ifdef _WIN32
int gc_GetFrame(LINEDEV linedev, GC_L2_BLK *l2_blkp);
#endif    /* _WIN32 */
int gc_GetLineDev(LINEDEV *linedevp, METAEVENT *metaeventp);
int gc_GetLinedevState(LINEDEV linedev, int type, int *state_buf);
#ifndef DOS
int gc_GetMetaEvent(METAEVENT *metaeventp);
#ifdef _WIN32
int gc_GetMetaEventEx(METAEVENT *metaeventp, unsigned long evt_handle );
#endif    /* _WIN32 */
#endif    /* !DOS */
int gc_GetNetworkH(LINEDEV linedev, int *networkhp);
int gc_GetParm(LINEDEV linedev, int parm_id, GC_PARM *valuep);
int gc_GetUsrAttr(LINEDEV linedev, void  **usr_attrp);
int gc_GetVer(LINEDEV linedev, unsigned int *releasenump,
         unsigned int *intnump, long component);
int gc_GetVoiceH(LINEDEV linedev, int * voicehp);
#ifdef _WIN32
int gc_HoldCall(CRN crn, unsigned long mode);
int gc_HoldCallACK(CRN crn);
int gc_HoldCallRej(CRN crn, int cause);
```

```
int gc_LibInit(void);
#endif    /* _WIN32 */
int gc_LoadDxParm(LINEDEV linedev, char *pathp, char *errmsgp, int err_length);
int gc_MakeCall(LINEDEV linedev, CRN *crnp, char *numberstr,
      GC_MAKECALL_BLK *makecallp, int timeout, unsigned long mode);
int gc_Open(LINEDEV *linedevp, char *devicename, int rfu);
int gc_OpenEx(LINEDEV *linedevp, char *devicename, int rfu, void *usrattr);

int gc_ReleaseCall(CRN crn);
int gc_ReqANI(CRN crn, char *anibuf, int req_type, unsigned long mode);
int gc_ResetLineDev(LINEDEV linedev, unsigned long mode);
int gc_ResultMsg(int cclibid, long result_code, char **msg);
int gc_ResultValue(METAEVENT *metaeventp, int *gc_result, int *cclibidp,
                   long *cclib_resultp);
#ifdef _WIN32
int gc_RetrieveCall(CRN crn, unsigned long mode);
int gc_RetrieveCallAck(CRN crn);
int gc_RetrieveCallRej(CRN crn, int cause);
#endif    /* _WIN32 */
int gc_SetBilling(CRN crn,int rate_type, GC_RATE_U *ratep, unsigned long mode);
int gc_SetCallingNum(LINEDEV linedev, char *calling_num);
int gc_SetChanState(LINEDEV linedev, int chanstate, unsigned long mode);
int gc_SetDlgerrValue(LINEDEV linedev, int dlgerr);
int gc_SetEvtMsk(LINEDEV linedev, unsigned long mask, int action);
int gc_SetInfoElem(LINEDEV linedev, GC_IE_BLK *iep);
int gc_SetParm(LINEDEV linedev, int parm_id, GC_PARM value);
int gc_SetUsrAttr(LINEDEV linedev, void *usr_attr);
#ifdef _WIN32
int gc_SndFrame(LINEDEV linedev, GC_L2_BLK* l2_blkp);
#endif    /* _WIN32 */
int gc_SndMsg(LINEDEV linedev, CRN crn, int msg_type, GC_IE_BLK *sndmsgptr);
int gc_Start(GC_START_STRUCT *startp);
int gc_StartTrace(LINEDEV linedev, char *tracefilename);
int gc_Stop(void);
int gc_StopTrace(LINEDEV linedev);
int gc_WaitCall(LINEDEV linedev, CRN *crnp, GC_WAITCALL_BLK *waitcallp,
     int timeout, unsigned long mode);
#if defined (__cplusplus)
}
#endif    /* __cplusplus */

#else     /* !_MSC_VER && !__STDC__ && !__cplusplus */
int gc_AcceptCall();
int gc_AnswerCall();
int gc_Attach();
int gc_CallAck();
int gc_CallProgress();
int gc_CCLibIDToName();
int gc_CCLibNameToID();
int gc_CCLibStatus();
int gc_CCLibStatusAll();
int gc_CRN2LineDev();
int gc_Close();
int gc_Detach();
int gc_DropCall();
int gc_ErrorValue();
int gc_ExtensionFunction();
int gc_GetANI();
int gc_GetBilling();
int gc_GetCallInfo();
int gc_GetCallState();
int gc_GetCRN();
```

```
int gc_GetDeviceNameInfo();
int gc_GetDlgerrValue();
int gc_GetDNIS();
#ifdef DOS
int gc_GetEvent();
#endif    /* DOS */
#ifdef _WIN32
int gc_GetFrame();
#endif    /* _WIN32 */
int gc_GetLineDev();
int gc_GetLinedevState();
#ifndef DOS
int gc_GetMetaEvent();
#ifdef _WIN32
int gc_GetMetaEventEx();
#endif    /* _WIN32 */
#endif    /* !DOS */
int gc_GetNetworkH();
int gc_GetParm();
int gc_GetUsrAttr();
int gc_GetVer();
int gc_GetVoiceH();
#ifdef _WIN32
int gc_HoldCall();
int gc_HoldCallACK();
int gc_HoldCallRej();
int gc_LibInit();
#endif    /* _WIN32 */
int gc_LoadDxParm();
int gc_MakeCall();
int gc_Open();
int gc_OpenEx();
int gc_ReleaseCall();
int gc_ReqANI();
int gc_ResetLineDev();
int gc_ResultMsg();
int gc_ResultValue();
#ifdef _WIN32
int gc_RetrieveCall();
int gc_RetrieveCallAck();
int gc_RetrieveCallRej();
#endif    /* _WIN32 */
int gc_SetBilling();
int gc_SetCallingNum();
int gc_SetChanState();
int gc_SetDlgerrValue();
int gc_SetEvtMsk();
int gc_SetInfoElem();
int gc_SetParm();
int gc_SetUsrAttr();
#ifdef _WIN32
int gc_SndFrame();
#endif    /* _WIN32 */
int gc_SndMsg();
int gc_Start();
int gc_StartTrace();
int gc_Stop();
int gc_StopTrace();
int gc_WaitCall();
#endif    /* _MSC_VER || __STDC__ || __cplusplus */

#ifdef _WIN32
```

```
#pragma pack()
#endif    /* _WIN32 */

#ifdef __cplusplus
}
#endif    /* __cplusplus */

#endif         /* _GCLIB_H */
```

gcerr.h Header File

```
/*****************************************************************
*
*    C Header:      gcerr.h
*    Instance:      dnj25
*    Description:   GlobalCall error header file for application use
*    %created_by:   wienerc %
*    %date_created: Mon Feb  9 13:09:58 1998 %
*
*****************************************************************/
/*****************************************************************
*  Copyright (C) 1996-1998 Dialogic Corp.
*  All Rights Reserved
*
*  THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF Dialogic Corp.
*  The copyright notice above does not evidence any actual or
*  intended publication of such source code.
*****************************************************************/

#ifndef _GCERR_H_
#define _GCERR_H_

#ifndef lint
static char    *_dnj25_gcerr_h = "@(#) %filespec: gcerr.h-11 %  (%full_filespec:
gcerr.h-11:incl:dnj25 %)";
#endif

/***************/
/* Error values */
/***************/
/*gc1*/
/* Note: when adding error codes, recall that ICAPI error codes for the
   most part may not excede 0x7F - cf. r2_updateline() */
#define GC_ERROR  -1
#define EGC_NOERR             0        /* No error                          */
#define GC_SUCCESS            EGC_NOERR /* synonym of EGC_NOERR             */
#define EGC_NOCALL            1        /* No call was made or transfered    */
#define EGC_ALARM             2        /* Function interrupt by alarm       */
#define EGC_ATTACHED          3        /* specified resource already attached  */
#define EGC_DEVICE            4        /* Bad device handle                 */
#define EGC_INVPROTOCOL       5        /* Ivalid protocol name              */
#define EGC_PROTOCOL          7        /* Protocol error                    */
#define EGC_SYNC              8        /* The mode flag must be EV_ASYNC    */
#define EGC_TIMEOUT           9        /* function time out                 */
#define EGC_UNSUPPORTED       0xA      /* Function is not supported         */
#define EGC_USER              0xB      /* Function interrupted by user      */
#define EGC_VOICE             0xc      /* No voice resource attached        */
#define EGC_NDEVICE           0xd      /* Too many devices opened           */
#define EGC_NPROTOCOL         0xe      /* Too many protocols opened         */
```

```
#define EGC_BADFCN            0xf     /* Bad function code (TSR)          */
#define EGC_TSRNOTACTIVE      0x10    /* CCLIB not active (TSR)           */
#define EGC_COMPATIBILITY     0x11    /* incompatible components          */
/*gc2*/
/*gc3*/
#define EGC_EVTERR            0x13    /* Internal Dialogic use only       */
/*gc4*/
#define EGC_PUTEVT            0x14    /* Error queuing event              */
#define EGC_MAXDEVICES        0x15    /* Exceeded Maximum devices limit   */
#define EGC_OPENH             0x16    /* Error opening voice channel      */
/*gc5*/
#define EGC_INTERR            0x18    /* Internal Global Call Error */
#define EGC_NOMEM             0x19    /* Out of memory                    */
#define EGC_PFILE             0x1A    /* Error opening parameter file     */
#define EGC_TIMER             0x1B    /* Error starting timer             */
#define EGC_FILEWRITE         0x1C    /* Error writing file               */
#define EGC_SYSTEM            0x1D    /* System error                     */
/*gc6*/
#define EGC_VOXERR            0x1E    /* Internal Dialogic use only       */
#define EGC_DTIERR            0x32    /* Internal Dialogic use only       */
/*gc7*/
#define EGC_ERR               0x39    /* Internal Dialogic use only       */
/*gc8*/
#define EGC_LINERELATED       0x40    /* Error is related to line device  */
#define EGC_INVSTATE          0x41    /* Invalid state                    */
#define EGC_INVCRN            0x42    /* Invalid call reference number    */
#define EGC_INVLINEDEV        0x43    /* Invalid line device passed       */
#define EGC_INVPARM           0x44    /* Invalid parameter(argument)      */
#define EGC_SRL               0x45    /* SRL failure                      */
/*gc9*/
#define EGC_OTHERERRORS       0x80    /* Internal Dialogic use only       */
#define EGC_USRATTRNOTSET     0x81    /* UsrAttr was not set for this ldev*/
#define EGC_INVMETAEVENT      0x82    /* Invalid metaevent                */
#define EGC_GCDBERR           0x83    /* GlobalCall database error        */
#define EGC_NAMENOTFOUND      0x84    /* trunk device name name not found */
#define EGC_DRIVER            0x85    /* driver error                     */
#define EGC_FILEREAD          0x86    /* File read                        */
#define EGC_FILEOPEN          0x87    /* file open                        */
#define EGC_TASKABORTED       0x88    /* task aborted                     */
#define EGC_CCLIBSPECIFIC     0x89    /* cclib specific - a catchall      */
#define EGC_XMITALRM          0x8A    /* Send alarm failed                */
#define EGC_SETALRM           0x8B    /* Set alarm mode failed            */
#define EGC_CCLIBSTART        0x8C    /* At least one cclib failed to start*/
#define EGC_ALARMDBINIT       0x8D    /* Alarm database failed to init    */
#define EGC_INVDEVNAME        0x8E    /* Invalid device name              */
#define EGC_DTOPEN            0x8F    /* dt open failed                   */
#define EGC_GCNOTSTARTED      0x90    /* GlobalCall not started           */
#define EGC_DUPENTRY          0x91    /* inserting a duplicate entry into
                                         the database */
#define EGC_ILLSTATE          0x92    /* Function is not supported in the
                                         current state */
#define EGC_FUNC_NOT_DEFINED  0x93    /* Low level function is not defined */
#define EGC_GC_STARTED        0x94    /* GlobalCall is already started    */
#define EGC_BUSY              0x95    /* Line is busy                     */
#define EGC_NOANSWER          0x96    /* Ring, no answer                  */
#define EGC_NOT_INSERVICE     0x97    /* Number not in service            */
#define EGC_NOVOICE           0x98    /* No voice                         */
#define EGC_NORB              0x99    /* no ringback                      */
#define EGC_CEPT              0x100   /* operator intercept               */
#define EGC_STOPD             0x101   /* call progress stopped            */
#define EGC_CPERROR           0x102   /* SIT detection error              */
#define EGC_DIALTONE          0x103   /* no dial tone detected            */
```

```
#define EGC_ROUTEFAIL         0x104    /* routing failed                  */
#define EGC_DTUNLISTEN        0x105    /* dt_unlisten failed              */
#define EGC_DXUNLISTEN        0x106    /* dx_unlisten failed              */
#define EGC_AGUNLISTEN        0x107    /* ag_unlisten failed              */
#define EGC_DTGETXMITSLOT     0x108    /* dt_getxmitslot failed           */
#define EGC_DXGETXMITSLOT     0x109    /* dx_getxmitslot failed           */
#define EGC_AGGETXMITSLOT     0x10A    /* ag_getxmitslot failed           */
#define EGC_DTLISTEN          0x10B    /* dt_listen failed                */
#define EGC_DXLISTEN          0x10C    /* dx_listen failed                */
#define EGC_AGLISTEN          0x10D    /* ag_listen failed                */

/* Note: when adding error codes, recall that ICAPI error codes for the
   most part may not excede 0x7F - cf. r2_updateline() */
/*****************/
/* result values */
/*****************/
/*gc9*/
#define GCRV_RESULT     0x500
#define GCRV_NORMAL     (GCRV_RESULT | 0)    /* normal completion             */
#define GCRV_ALARM      (GCRV_RESULT | 1)    /* event caused by alarm         */
#define GCRV_TIMEOUT    (GCRV_RESULT | 2)    /* event caused by timeout       */
#define GCRV_PROTOCOL   (GCRV_RESULT | 3)    /* event caused by protocol error*/
#define GCRV_NOANSWER   (GCRV_RESULT | 4)    /* event caused by no answer     */
#define GCRV_INTERNAL   (GCRV_RESULT | 5)    /* event caused internal failure */
#define GCRV_CCLIBSPECIFIC (GCRV_RESULT | 6) /* event caused by cclib specific
                                                failure */
#define GCRV_NOVOICE    (GCRV_RESULT | 7)    /* Call needs voice, use ic_attach()
                                                */
#define GCRV_SIGNALLING (GCRV_RESULT | 8)    /* Signaling change */
#define GCRV_BUSY       (GCRV_RESULT | 9)    /* Line is busy */
#define GCRV_NOT_INSERVICE (GCRV_RESULT | 0x40) /* Number not in service     */
#define GCRV_NORB       (GCRV_RESULT | 0x41) /* no ringback           */
#define GCRV_CEPT       (GCRV_RESULT | 0x42) /* operator intercept    */
#define GCRV_STOPD      (GCRV_RESULT | 0x43) /* call progress stopped */
#define GCRV_CPERROR    (GCRV_RESULT | 0x44) /* call progress error   */
#define GCRV_DIALTONE   (GCRV_RESULT | 0x45) /* no dial tone          */

/*
-- alarm values
-- initialized such that matches values (well actually +0x10)
-- in DTT1_xxx and DTE1_xxx (as of 4/8/96).
-- Also, doesn't differentiate between T1 & E1 when the same value
-- is used for both with a different meaning. The same value
-- will be returned here (albeit from different mneumonics)
-- User is expected to know if the board is T1 or E1
-- At the present time (4/8/96) Dialogic software does not tell
-- the difference at the library level.
-- gc_ResultMsg() will use the E1 vocabulary.
*/
#define GCRV_OOF     (GCRV_RESULT | 0x10) /* out of frame error, count saturation
                                             */
#define GCRV_LOS     (GCRV_RESULT | 0x11) /* Initial loss of signal detection */
#define GCRV_DPM     (GCRV_RESULT | 0x12) /* Driver performance monitor failure
                                             */
#define GCRV_BPVS    (GCRV_RESULT | 0x13) /* Bipolar violation count saturation
                                             */
#define GCRV_ECS     (GCRV_RESULT | 0x14) /* Error count saturation */
#define GCRV_RYEL    (GCRV_RESULT | 0x15) /* Received yellow alarm */
#define GCRV_RRA     GCRV_RYEL            /* Received remote alarm */
#define GCRV_RCL     (GCRV_RESULT | 0x16) /* Received carrier loss */
#define GCRV_FERR    (GCRV_RESULT | 0x17) /* Frame bit error */
#define GCRV_FSERR   GCRV_FERR            /* Received frame sync error */
```

```
#define GCRV_B8ZSD   (GCRV_RESULT | 0x18) /* Bipolar eight zero substitution
                                             detect */
#define GCRV_RBL     (GCRV_RESULT | 0x19) /* Received blue alarm */
#define GCRV_RUA1    GCRV_RBL             /* Received unframed all 1s */
#define GCRV_RLOS    (GCRV_RESULT | 0x1A) /* Received loss of sync */
#define GCRV_RED     (GCRV_RESULT | 0x1B) /* Got a read alarm condition */
#define GCRV_MFSERR  (GCRV_RESULT | 0x1C) /* Received multi frame sync error */
#define GCRV_RSA1    (GCRV_RESULT | 0x1D) /* Received signalling all 1s */
#define GCRV_RDMA    (GCRV_RESULT | 0x1E) /* Received distant multi-frame alarm
                                              */
#define GCRV_CECS    (GCRV_RESULT | 0x1F) /* CRC4 error count saturation */

/* -- recovered series -- */
#define GCRV_OOFOK   (GCRV_RESULT | 0x20) /* out of frame error, count saturation
                                             recovered */
#define GCRV_LOSOK   (GCRV_RESULT | 0x21) /* Initial loss of signal detection
                                             recovered */
#define GCRV_DPMOK   (GCRV_RESULT | 0x22) /* Driver performance monitor failure
                                             recovered */
#define GCRV_BPVSOK  (GCRV_RESULT | 0x23) /* Bipolar violation count saturation
                                             recovered */
#define GCRV_ECSOK   (GCRV_RESULT | 0x24) /* Error count saturation recovered */
#define GCRV_RYELOK  (GCRV_RESULT | 0x25) /* Received yellow alarm recovered */
#define GCRV_RRAOK   GCRV_RYELOK          /* Received remote alarm recovered */
#define GCRV_RCLOK   (GCRV_RESULT | 0x26) /* Received carrier loss recovered */
#define GCRV_FERROK  (GCRV_RESULT | 0x27) /* Frame bit error recovered */
#define GCRV_FSERROK GCRV_FERROK          /* Received frame sync error recovered
                                             */
#define GCRV_B8ZSDOK (GCRV_RESULT | 0x28) /* Bipolar eight zero substitution dtct
                                             recovered */
#define GCRV_RBLOK   (GCRV_RESULT | 0x29) /* Received blue alarm recovered */
#define GCRV_RUA1OK  GCRV_RBLOK           /* Received unframed all 1s recovered
                                              */
#define GCRV_RLOSOK  (GCRV_RESULT | 0x2A) /* Received loss of sync recovered */
#define GCRV_REDOK   (GCRV_RESULT | 0x2B) /* Got a read alarm condition recovered
                                              */
#define GCRV_MFSERROK (GCRV_RESULT | 0x2C) /* Received multi frame sync error
                                               recovered */
#define GCRV_RSA1OK  (GCRV_RESULT | 0x2D) /* Received signalling all 1s recovered
                                              */
#define GCRV_RDMAOK  (GCRV_RESULT | 0x2E) /* Received distant multi-frame alarm
                                             recovered */
#define GCRV_CECSOK  (GCRV_RESULT | 0x2F) /* CRC4 error count saturation
                                             recovered */

#endif    /* _GCERR_H_ */
```

Appendix D
Related Publications

This appendix lists publications you should refer to for additional information on Dialogic products or communications technology.

Dialogic Hardware References

- *Quick Installation Card* for your boards

Dialogic Software References

- *GlobalCall Analog Technology User's Guide for UNIX and Windows NT*
- *GlobalCall E-1/T-1 Technology User's Guide for UNIX and Windows NT*
- *GlobalCall Country Dependent Parameters (CDP) Reference*
- *GlobalCall ISDN Technology User's Guide for UNIX and Windows NT*
- *System Software Release documentationr later for UNIX*
- *System Release documentation for your operating system*
- *SCbus Routing Guide*
- *SCbus Routing Function Reference for UNIX*
- *SCbus Routing Function Reference for Windows NT*
- *Voice Software Reference and Standard Runtime Library Programmer's Guide for UNIX*
- *Voice Software Reference for Windows NT and Standard Runtime Library Programmer's Guide for Windows NT*

Communications Technology References

- Edgar, Bob, *PC Based Voice Processing*, New York: Flatiron Publishing Inc. 2nd edition, 1994, ISBN 0-936648-47-7
- Newton, Harry, *Newton's Telecom Dictionary* (12th edition), Flatiron Publishing, Inc. 1997, ISBN 1-57820-008-3

R2 MF Signaling References

- *Specifications of Signaling Systems R1 and R2*, International Telegraph and Telephone Consultative Committee (CCITT), Blue Book Vol. VI, Fascicle VI.4, ISBN 92-61-03481-0
- *General Recommendations on Telephone Switching and Signaling*, International Telegraph and Telephone Consultative Committee (CCITT), Blue Book Vol. VI, Fascicle VI.1, ISBN 92-61-03451-9

ISDN Signaling References

- CCITT. *CCITT Recommendation Digital Subscriber Signalling System No. 1 (DSS 1), Network Layer, User-Network Management, Vol. VI - Fascicle VI.11, Rec. Q.930 - Q.940*. Geneva: CCITT, 1989.
- Stallings, William. *ISDN: An Introduction*. New York: Macmillan Publishing Company, 1992.

T-1 Robbed Bit Signaling References

- Bellamy, John, *Digital Telephony*, 2nd ed. New York: John wiley & Sons, 1991
- Fike, John L., and George Friend, *Understanding Telephone Electronics*, Indiana: Howard W. Sams & Company, 1988
- Flanagan, William A., *The Guide to T-1 Networking*, 4th ed. New York, Telecom Library Inc., 1990
- *LATA Switching Systems Generic Requirements (LSSGR)*, Bellcore Technical Reference TR-TSY-000064, Issue 2, July 1987, and modules, Bellcore

Glossary

analog: 1. Refers to the telephone line interface that receives analog voice and telephony signaling information from the telephone network. 2. Refers to applications that use loop start signaling instead of digital signaling. 3. A method of telephony transmission in which the information from the source (for example, speech in a human conversation) is converted into an electrical signal that varies continuously over a range of amplitude values. 4. Telephone transmissions or switching that is not digital. See also *ground start, loop start*.

analog interface: see *analog, loop start*.

analog loop start: see *analog, loop start*.

analog voice: see *analog, loop start*.

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.

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.

atomic synchronous function: typically terminates immediately, returns control to the application and does not cause a call state transition.

available library: A call control library configured to be recognized by the GlobalCall API and successfully started by the GlobalCall **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/sec in either direction

blind dialing: Dialing without waiting for dial tone detection.

bonding: Bandwidth ON Demand INteroperability Group - an inverse-multiplexing method used to combine multiple channels into a single, coherent channel.

BRI: Basic Rate Interface - interface for connecting data terminal and voice telephones to an ISDN switch. The BRI includes two 64 Kbps B channels and one 16 Kbps D channel.

call analysis: a process used to automatically determine what happened after an outbound call is dialed. Call analysis monitors the progress of an outbound call after dialing and provides information to allow the application to process the call based on the status of the call. Call analysis can determine 1) if the line is answered and, in many cases, how the line is answered, 2) if the line rings but is not answered, 3) if the line is busy or 4) the problem in completing the call. Also referred to as 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 GlobalCall functions to implement network specific commands and communications.

call progress tone: a tone sent from the PTT to tell the calling party the progress of the call, (e.g., a dial tone, busy tone, ringback tone, etc.). 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 GlobalCall 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 E-1 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 T-1 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 Adminstrations Europeenes des Postes et Telecommunications. A collection of groups that set European telecommunications standards.

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

configured library: A call control library supported by the GlobalCall API.

congestion: Flow of user-to-user data

CRN - see Call Reference Number.

D channel: The data channel in an ISDN interface that carries control signals and customer call data in packets.

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

DLL (Dynamically Linked Library) (Windows NT)**:** 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 *time slot*.

digital channel: Designates a bi-directional transfer of data for a single time slot of a T-1 or an E-1 digital frame between a T-1/E-1 device that connects to the digital service and the SCbus. Digitized information from the T-1/E-1 device is sent to the SCbus over the digital transmit channel. The response to this call is sent from the SCbus to the T-1/E-1 device over the digital receive (listen) channel.

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

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.

DPNSS Digital Private Network Signaling System. An E-1 primary rate protocol used in Europe to pass calls transparently between PBXs.

E-1 CAS: E-1 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.

E-1: 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).

E&M: In an analog environment, an electrical circuit containing separate signaling leads in addition to the leads for receiving and transmitting audio. There can be a total of 4 or 6 wires, referred to as “four wire E&M” and “six wire E&M”. In addition to the audio pairs, a pair of dedicated signaling leads called “Ear” and “Mouth” exist. See also *analog*, *loop start*.

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: For Windows NT 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 non-stub call control library configured to be recognized by the GlobalCall API and which did not successfully start when the GlobalCall **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.

ground start: In an analog environment, an electrical circuit consisting of 2 wires (or leads) called tip and ring, which are the 2 conductors of a telephone cable pair. The CO provides voltage (called "talk battery" or just "battery") to power the line. Although this sounds like loop start, the difference is in the way the phone line is "siezed," or how the originator of the call signals the CO. When using Dialogic equipment, an application cannot originate a call on a ground start line. However, Dialogic equipment can receive and process calls (transfer, for example) on ground start lines. See also *analog, loop start*.

ICAPI: The Dialogic Interface Control Application Programming Interface, which provides a device specific telephony and signaling interface for the GlobalCall API to control Dialogic network interface boards using T-1 robbed bit or E-1 CAS signaling schemes. Also the name of a call control library configured for GlobalCall.

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.

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 GlobalCall.

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

loop start: In an analog environment, an electrical circuit consisting of 2 wires (or leads) called tip and ring, which are the 2 conductors of a telephone cable pair. The CO provides voltage (called "talk battery" or just "battery") to power the line. When the circuit is complete, this voltage produces a current called loop current. The circuit provides a method of starting (seizing) a telephone line or trunk by sending a

supervisory signal (going off-hook) to the CO. See also *analog, ground start*.

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. When using the MS-DOS operating system, GlobalCall multitasking functions operate in the same manner as asynchronous functions.

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 analog (loop start, ground start, etc.) and digital (E-1 CAS, T-1 robbed bit, and ISDN) network interface devices. Network resources are assigned to telephone lines (i.e., 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.

NFAS: Network Facility Associated Signal - allows multiple spans to be controlled by a single D channel subaddressing.

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. UNIX and Windows NT use preemptive multitasking to support multiple simultaneous processes.

PRI: Primary Rate Interface - interface at the ends of high-volume trunks linking CO facilities and ISDN network switches to each other. A T-1 ISDN PRI transmits 23 B channels and one D channel, each at 64 Kbps. An E-1 ISDN PRI transmits 30 B channels, one D channel and one framing channel, each at 64 Kbps.

primary thread: *see thread*.

process (UNIX): the execution of a program. In UNIX, 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 NT): (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 NT 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 NT actually schedules for execution. When an application is launched, Windows NT starts a process and a primary thread.

Windows NT processes:

1. are implemented as objects and accessed using object services;
2. can have multiple threads executing in their address space;
3. have build-in synchronization for both process objects and thread objects.

In contrast to UNIX, Windows NT does not use a parent/child relationship with the processes it creates.

Process or System Scheduler for UNIX: 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. See also *synchronization objects* for Windows NT.

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 TDM (Time Division Multiplexed) resource sharing bus that allows information to be transmitted and received among resources over multiple data lines. A hardwired connection between Switch Handlers (SC2000 chips) 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.

SIT - Special Information Tone

Special Information Tone (SIT)

SpringBoard: A Dialogic expansion board using digital signal processing to emulate the functions of other products.

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.

stub library: A library with a minimal set of internal functions that represents a call control library that is not required for a particular application. This stub library is entered into the list of configured call control libraries recognized by the GlobalCall API but is not capable of being started. (Used only to avoid link errors.)

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 NT 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 NT 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. See also *Process Scheduler* for UNIX.

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.

T-1: 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. T-1 is a digital transmission standard in North America.

T-1 robbed bit: A T-1 digital line using robbed bit signaling. In T-1 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.

termination events: GlobalCall events returned to the application to terminate function calls.

thread (Windows NT): 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.

TSR: Transmit and Stay Resident. Loading a program into memory in a MS-DOS operating system so that the program is always ready to run.

unsolicited event: an event that occurs without prompting (e.g., GCEV_BLOCKED, GCEV_UNBLOCKED, etc.).

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

voice channel: Designates a bi-directional transfer of data for a single call between a voice device processing that call and the SCbus. Digitized voice from the analog or T-1/E-1 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 analog or T-1/E-1 interface device.

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

voice resource: same as a voice channel.

Index

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Y

NOTES

NOTES

NOTES