



# Global Call API for Linux and Windows Operating Systems

Library Reference

---

*February 2003*



INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

This Global Call API for Linux and Windows Operating Systems Library Reference as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without express written consent of Intel Corporation.

Copyright © 1996-2003 Intel Corporation. All Rights Reserved.

AnyPoint, AppChoice, BoardWatch, BunnyPeople, CablePort, Celeron, Chips, CT Media, Dialogic, DM3, EtherExpress, ETOX, FlashFile, i386, i486, i960, iCOMP, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Create&Share, Intel GigaBlade, Intel InBusiness, Intel Inside, Intel Inside logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel Play, Intel Play logo, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel TeamStation, Intel Xeon, Intel XScale, IPLink, Itanium, MCS, MMX, MMX logo, Optimizer logo, OverDrive, Paragon, PC Dads, PC Parents, PDCharm, Pentium, Pentium II Xeon, Pentium III Xeon, Performance at Your Command, RemoteExpress, SmartDie, Solutions960, Sound Mark, StorageExpress, The Computer Inside., The Journey Inside, TokenExpress, Trillium, VoiceBrick, VTune, and Xircom are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

\* Other names and brands may be claimed as the property of others.

Publication Date: February 2003

Document Number: 05-1816-003

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

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

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

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



# Contents

---

<b>About This Publication</b> .....	9
Purpose .....	9
Intended Audience .....	9
How to Use This Publication .....	9
Related Information .....	10
<b>1 Function Summary by Category</b> .....	11
1.1 Global Call Basic Functions .....	11
1.2 Library Information Functions .....	12
1.3 Optional Call Handling Functions .....	12
1.4 Advanced Call Model Functions .....	13
1.5 System Controls and Tools Functions .....	13
1.6 Voice and Media Functions .....	15
1.7 ISDN Interface Specific Functions .....	15
1.8 Global Call Alarm Management System (GCAMS) Functions .....	16
1.9 Feature Transparency and Extension (FTE) Functions .....	18
1.10 Real Time Configuration Management (RTCM) Functions .....	18
1.11 Global Call Service Request (GCSR) Functions .....	18
1.12 GC_PARM_BLK Utility Functions .....	18
1.13 Deprecated Functions .....	19
1.14 Global Call Function Support by Technology .....	20
<b>2 Function Information</b> .....	27
2.1 Function Syntax Conventions .....	27
gc_AcceptCall() – indicate to originator that call will be answered .....	28
gc_AlarmName() – return the name of the alarm for the current alarm event .....	31
gc_AlarmNumber() – return the alarm number for the current alarm event .....	33
gc_AlarmNumberToName() – convert an alarm number to a name .....	35
gc_AlarmSourceObjectID() – return the ASO ID for the current alarm event .....	37
gc_AlarmSourceObjectIDToName() – convert the ASO ID to the ASO name .....	39
gc_AlarmSourceObjectName() – return the ASO name of the current alarm .....	41
gc_AlarmSourceObjectNameToID() – convert the ASO name to the ASO ID .....	43
gc_AnswerCall() – indicate to originator that destination party is connected .....	45
gc_Attach() – attach a voice resource to a line device .....	48
gc_AttachResource() – attach a voice or media resource to a line device .....	49
gc_BlindTransfer() – initiate and complete a one-step transfer .....	53
gc_CallAck() – indicate (to the originator) call reception .....	56
gc_CallProgress() – provide information about the progress of a call .....	60
gc_CCLibIDToName() – convert call control library ID to name .....	63
gc_CCLibNameToID() – convert call control library name to ID .....	65
gc_CCLibStatus() – retrieve the status of a single call control library .....	67
gc_CCLibStatusAll() – retrieve the status of all call control libraries .....	69
gc_CCLibStatusEx() – retrieve call control library status .....	71

gc_Close( ) – close a previously opened device . . . . .	74
gc_CompleteTransfer( ) – complete the transfer of a call . . . . .	76
gc_CRN2LineDev( ) – map a CRN to its line device ID . . . . .	79
gc_Detach( ) – detach a voice or media resource from the line device. . . . .	81
gc_DropCall( ) – disconnect a call. . . . .	84
gc_ErrorInfo( ) – provide error information about a failed function . . . . .	87
gc_ErrorValue( ) – get an error value/failure reason code . . . . .	89
gc_Extension( ) – provide a generic interface for technology-specific features . . . . .	91
gc_GetAlarmConfiguration( ) – retrieve alarm configuration parameter values . . . . .	95
gc_GetAlarmFlow( ) – indicate which alarms are sent to the application . . . . .	101
gc_GetAlarmParm( ) – retrieve parameter data for a parameter set ID . . . . .	104
gc_GetAlarmSourceObjectList( ) – retrieve a list of all alarm source objects . . . . .	107
gc_GetAlarmSourceObjectNetworkID( ) – retrieve the ID of the layer 1 ASO. . . . .	110
gc_GetANI( ) – retrieve ANI information . . . . .	112
gc_GetBilling( ) – retrieve the billing information. . . . .	114
gc_GetCallInfo( ) – retrieve information associated with the call. . . . .	116
gc_GetCallProgressParm( ) – retrieve protocol call progress parameters . . . . .	121
gc_GetCallState( ) – retrieve the state of the call . . . . .	123
gc_GetConfigData( ) – retrieve parameter values for a given target object . . . . .	127
gc_GetCRN( ) – retrieve a call reference number. . . . .	134
gc_GetCTInfo( ) – retrieve CT Bus time slot information. . . . .	136
gc_GetDNIS( ) – retrieve the DNIS information. . . . .	138
gc_GetFrame( ) – retrieve a Layer 2 frame. . . . .	141
gc_GetInfoElem( ) – retrieve IEs associated with a line device. . . . .	144
gc_GetLineDev( ) – retrieve a line device associated with an event. . . . .	147
gc_GetLinedevState( ) – retrieve the status of the line device . . . . .	149
gc_GetMetaEvent( ) – retrieve event information for the current SRL event. . . . .	152
gc_GetMetaEventEx( ) – retrieve event information for an SRL event . . . . .	158
gc_GetNetCRV( ) – retrieve the network call reference value. . . . .	161
gc_GetNetworkH( ) – retrieve the network device handle. . . . .	164
gc_GetParm( ) – retrieve the value of the specified parameter. . . . .	166
gc_GetResourceH( ) – retrieve the resource device handle . . . . .	168
gc_GetSigInfo( ) – retrieve the signaling information of an incoming message . . . . .	170
gc_GetUserInfo( ) – retrieve technology-specific user information . . . . .	174
gc_GetUsrAttr( ) – retrieve the user-defined attribute . . . . .	176
gc_GetVer( ) – retrieve the version number of a specified component. . . . .	178
gc_GetVoiceH( ) – retrieve the voice device handle . . . . .	182
gc_GetXmitSlot( ) – retrieve the network CT Bus time slot number . . . . .	184
gc_HoldACK( ) – accept a hold request from remote equipment . . . . .	187
gc_HoldCall( ) – place an active call on hold. . . . .	190
gc_HoldRej( ) – reject a hold request from remote equipment . . . . .	193
gc_LinedevToCCLIBID( ) – retrieve ID of call control library that opened a device . . . . .	196
gc_Listen( ) – connect a channel to a network CT Bus time slot. . . . .	198
gc_LoadDxParm( ) – set voice parameters associated with a line device . . . . .	201
gc_MakeCall( ) – make an outgoing call . . . . .	208
gc_Open( ) – open a Global Call device . . . . .	213

gc_OpenEx( ) – open a Global Call device and set a user-defined attribute . . . . .	214
gc_QueryConfigData( ) – query the configuration data . . . . .	221
gc_ReleaseCall( ) – release the call and the associated internal resources . . . . .	226
gc_ReleaseCallEx( ) – release the call and the associated internal resources . . . . .	227
gc_ReqANI( ) – request the remote side to return ANI. . . . .	229
gc_ReqMoreInfo( ) – request more information such as ANI or DNIS . . . . .	232
gc_ReqService( ) – request a service from a remote device . . . . .	236
gc_ResetLineDev( ) – reset the line device state and disconnect calls . . . . .	239
gc_RespService( ) – generate a response to a requested service. . . . .	242
gc_ResultInfo( ) – retrieve information about Global Call events . . . . .	245
gc_ResultMsg( ) – retrieve an ASCII string describing a result code . . . . .	247
gc_ResultValue( ) – retrieve the cause of an event . . . . .	249
gc_RetrieveAck( ) – accept a retrieve request from remote equipment . . . . .	251
gc_RetrieveCall( ) – retrieve a call from the OnHold state . . . . .	254
gc_RetrieveRej( ) – reject a retrieve request from remote equipment . . . . .	257
gc_SendMoreInfo( ) – send more information to the remote side . . . . .	260
gc_SetAlarmConfiguration( ) – set alarm configuration parameter values . . . . .	263
gc_SetAlarmFlow( ) – configure which alarms are sent to the application. . . . .	269
gc_SetAlarmNotifyAll( ) – set the notification attribute of all alarms . . . . .	272
gc_SetAlarmParm( ) – set the data associated with the alarm parameter . . . . .	275
gc_SetBilling( ) – set billing information . . . . .	278
gc_SetCallingNum( ) – set the default calling party number . . . . .	281
gc_SetCallProgressParm( ) – override default call progress parameters . . . . .	283
gc_SetChanState( ) – set the channel state of the indicated channel . . . . .	285
gc_SetConfigData( ) – update the configuration data . . . . .	288
gc_SetEvtMsk( ) – set the event mask associated with a specified line device . . . . .	294
gc_SetInfoElem( ) – set an additional information element . . . . .	298
gc_SetParm( ) – set the default parameters. . . . .	300
gc_SetUpTransfer( ) – initiate a supervised call transfer . . . . .	303
gc_SetUserInfo( ) – permit the setting of technology-specific user information . . . . .	306
gc_SetUsrAttr( ) – set an attribute defined by the user . . . . .	309
gc_SndFrame( ) – send a Layer 2 frame . . . . .	311
gc_SndMsg( ) – send non-call state related ISDN messages . . . . .	314
gc_Start( ) – start and initialize call control libraries . . . . .	317
gc_StartTrace( ) – start logging debug information . . . . .	320
gc_Stop( ) – stop call control libraries and release resources . . . . .	322
gc_StopTrace( ) – stop logging debug information. . . . .	324
gc_StopTransmitAlarms( ) – stop the transmission of one or more alarms . . . . .	326
gc_SwapHold( ) – switch between an active call and a call on hold . . . . .	329
gc_TransmitAlarms( ) – start the transmission of alarms . . . . .	332
gc_UnListen( ) – disconnect a channel from the network CT Bus time slot . . . . .	335
gc_util_delete_parm_blk( ) – delete the specified GC_PARM_BLK. . . . .	337
gc_util_find_parm( ) – find a parameter in a GC_PARM_BLK. . . . .	339
gc_util_insert_parm_ref( ) – insert a parameter by reference into a GC_PARM_BLK. . . . .	341
gc_util_insert_parm_val( ) – insert a parameter by value into a GC_PARM_BLK. . . . .	344
gc_util_next_parm( ) – retrieve the next parameter in a GC_PARM_BLK . . . . .	347

	gc_WaitCall( ) – indicate that the application is ready to receive inbound calls . . . . .	349
<b>3</b>	<b>Events</b> . . . . .	<b>353</b>
3.1	Event Types . . . . .	353
3.2	Event Information . . . . .	353
<b>4</b>	<b>Data Structures</b> . . . . .	<b>365</b>
	ALARM_FIELD – information about an alarm . . . . .	367
	ALARM_LIST – list of alarms for an ASO . . . . .	368
	ALARM_PARM_FIELD – data for an alarm parameter. . . . .	369
	ALARM_PARM_LIST – list of alarm parameters and all fields . . . . .	370
	ALARM_SOURCE_OBJECT_FIELD – entry in ALARM_SOURCE_OBJECT_LIST . . . . .	371
	ALARM_SOURCE_OBJECT_LIST – information about ASOs. . . . .	372
	CCLIB_START_STRUCT – startup information for a call control library. . . . .	373
	CT_DEVINFO – information about a Global Call line device . . . . .	374
	DX_CAP – call progress information for a Global Call line device . . . . .	375
	EXTENSIONEVTBLK – technology-specific information. . . . .	380
	GC_CALLACK_BLK – information for gc_CallAck( ) . . . . .	381
	GC_CCLIB_STATE – status of a call control library . . . . .	383
	GC_CCLIB_STATUS – states of a call control library. . . . .	384
	GC_CCLIB_STATUSALL – status of all call control libraries . . . . .	385
	GC_CUSTOMLIB_STRUCT – custom library information . . . . .	386
	GC_IE_BLK – used to send an IE block . . . . .	387
	GC_INFO – error or result information . . . . .	388
	GC_L2_BLK – used to send and receive layer 2 information . . . . .	389
	GC_MAKECALL_BLK – information for gc_MakeCall( ) . . . . .	390
	GC_PARM – union of data types . . . . .	391
	GC_PARM_BLK – parameter data . . . . .	392
	GC_PARM_DATA – parameter data. . . . .	393
	GC_PARM_ID – configuration information returned by gc_QueryConfigData( ) . . . . .	394
	GC_RATE_U – used to set billing rates for Vari-A-Bill service . . . . .	395
	GC_RTCM_EVTDATA – information returned via RTCM events . . . . .	396
	GC_START_STRUCT – specify which call control libraries are to be started . . . . .	397
	GCLIB_ADDRESS_BLK – called party or calling party address information . . . . .	398
	GCLIB_CALL_BLK – call information . . . . .	400
	GCLIB_CHAN_BLK – channel information . . . . .	401
	GCLIB_MAKECALL_BLK – generic call related parameters . . . . .	402
	METAEVENT – event descriptor for a metaevent. . . . .	403
	SC_TSINFO – CT Bus time slot information. . . . .	405
<b>5</b>	<b>Error Codes</b> . . . . .	<b>407</b>
<b>6</b>	<b>Supplementary Reference Information</b> . . . . .	<b>417</b>
6.1	Alarm Source Object IDs . . . . .	417
6.2	Target Objects . . . . .	417
	<b>Index</b> . . . . .	<b>425</b>

## *Figures*

---

1	Component Version Number Format. ....	179
2	GC_PARM_BLK Memory Diagram . ....	419
3	Sample GC_PARM_BLK Memory Diagram . ....	420

# Tables

---

1	Global Call Function Support by Technology . . . . .	20
2	Possible Values for the type Field in GC_CALLACK_BLK . . . . .	57
3	Fields in the 'info' Structure for GCACK_SERVICE_INFO . . . . .	57
4	gc_DropCall( ) Causes . . . . .	85
5	Alarm Configuration Types . . . . .	96
6	gc_GetCallInfo( ) info_id Parameter ID Definitions . . . . .	116
7	gc_GetSigInfo( ) info_id Parameter ID Definitions . . . . .	171
8	gc_GetVer( ) Return Values . . . . .	180
9	Call Conditions and Results . . . . .	210
10	Query IDs Defined in GCLib . . . . .	222
11	Result Values for GCEV_MOREINFO . . . . .	233
12	Possible Scope Settings for the alarm_config_type Parameter . . . . .	265
13	Possible Flag Settings for value Parameter . . . . .	273
14	Parameter Configuration Data Conditions and Results . . . . .	290
15	mask Parameter Values . . . . .	295
16	Parameter Descriptions, gc_GetParm( ) and gc_SetParm( ) . . . . .	301
17	Service Type Data Structure Field Descriptions . . . . .	382
18	Alarm Source Object IDs . . . . .	417
19	Supported Target Types . . . . .	418
20	Target Type and Target ID Pairs . . . . .	418
21	Possible Set ID, Parm ID Pairs used in GCLIB_MAKECALL_BLK Structure . . . . .	421
22	GC Parameter Entry List Maintained in GCLIB . . . . .	422
23	Examples of Parameter Entry List Maintained in CCLIB . . . . .	424





## About This Publication

---

The following topics provide information about this publication:

- [Purpose](#)
- [Intended Audience](#)
- [How to Use This Publication](#)
- [Related Information](#)

### Purpose

This publication provides a reference to all functions, events, data structures, and error codes in the Global Call API library. Supplemental information about using the Global Call API with specific technologies such as analog, E-/T-1, IP, ISDN, and SS7 is provided in the *Global Call Technology User's Guides*.

This publication is a companion document to the *Global Call API Programming Guide*, which provides guidelines for developing applications using the Global Call API.

### Intended Audience

This information is intended for:

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

### How to Use This Publication

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

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

The information in this publication is organized as follows:

- [Chapter 1, “Function Summary by Category”](#) introduces you to the various categories of functions in the Global Call API library.
- [Chapter 2, “Function Information”](#) provides an alphabetical reference to the Global Call API functions.
- [Chapter 3, “Events”](#) provides an alphabetical reference to the events used by the Global Call software.
- [Chapter 4, “Data Structures”](#) provides an alphabetical reference to the data structures used by the functions in the Global Call API library.
- [Chapter 5, “Error Codes”](#) describes the error codes available in the Global Call software.
- [Chapter 6, “Supplementary Reference Information”](#) provides reference information about:
  - Alarm Source Object (ASO) IDs
  - Target objects

## Related Information

See the following for more information:

- *Global Call API Programming Guide* – provides guidelines for developing applications using the Global Call API
- *Global Call Technology User’s Guides* – provide information about using the Global Call API with specific technologies:
  - *GlobalCall Analog Technology User’s Guide*
  - *GlobalCall E-1/T-1 CAS/R2 Technology User’s Guide*
  - *Global Call IP over Embedded Stack Technology User’s Guide*
  - *Global Call IP over Host-based Stack Technology User’s Guide*
  - *Global Call ISDN Technology User’s Guide*
  - *Global Call SS7 Technology User’s Guide*
- *Standard Runtime Library API Programming Guide* and *Standard Runtime Library API Library Reference* – describe the Standard Runtime Library (SRL), a device-independent library that consists of event management functions and standard attribute functions. The Global Call API is one of a family of APIs that use the SRL to deliver response events to API commands.
- Release Guide for your system release – provides information about the system release, system requirements, software and hardware features, supported hardware, and release documentation
- Release Update for your system release (available on the Technical Support Web site only) – describes compatibility issues, restrictions and limitations, known problems, and late-breaking updates or corrections to the release documentation
- <http://developer.intel.com/design/telecom/support/> – Technical Support Web site which contains developer support information, downloads, release documentation, technical notes, application notes, a user discussion forum, and more

This chapter describes the categories into which the Global Call API library functions can be logically grouped. It also includes a table showing which technologies (T-1/E-1, ISDN, IP, etc.) are supported for each of the functions. The topics in this chapter are:

• Global Call Basic Functions .....	11
• Library Information Functions .....	12
• Optional Call Handling Functions .....	12
• Advanced Call Model Functions .....	13
• System Controls and Tools Functions .....	13
• Voice and Media Functions .....	15
• ISDN Interface Specific Functions .....	15
• Global Call Alarm Management System (GCAMS) Functions .....	16
• Feature Transparency and Extension (FTE) Functions .....	18
• Real Time Configuration Management (RTCM) Functions .....	18
• Global Call Service Request (GCSR) Functions .....	18
• GC_PARM_BLK Utility Functions .....	18
• Deprecated Functions .....	19
• Global Call Function Support by Technology .....	20

## 1.1 Global Call Basic Functions

The following functions are used to interface with all signaling systems. See the *Global Call API Programming Guide* for more information about the basic call state model.

### **gc\_AnswerCall()**

indicates that the destination party is connected

### **gc\_DropCall()**

disconnects a call

### **gc\_MakeCall()**

makes an outgoing call

### **gc\_ReleaseCall()**

deprecated: use **gc\_ReleaseCallEx()**

### **gc\_ReleaseCallEx()**

releases the call and the associated internal resources

**gc\_WaitCall()**

indicates that the application is ready to receive inbound calls

## 1.2 Library Information Functions

The following functions retrieve the status, names, and numbers of call control libraries.

**gc\_CCLibIDToName()**

converts call control library ID to name

**gc\_CCLibNameToID()**

converts call control library name to ID

**gc\_CCLibStatus()**

deprecated: use **gc\_CCLibStatusEx()**

**gc\_CCLibStatusAll()**

deprecated: use **gc\_CCLibStatusEx()**

**gc\_CCLibStatusEx()**

retrieves call control library status

**gc\_LinedevToCCLIBID()**

retrieves ID of call control library that opened a device

## 1.3 Optional Call Handling Functions

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

**gc\_AcceptCall()**

indicates to the originator that the call will be answered

**gc\_CallAck()**

indicates call reception and optionally takes action

**gc\_GetANI()**

deprecated: use **gc\_GetCallInfo()**

**gc\_GetBilling()**

retrieves billing information

**gc\_GetDNIS()**

deprecated: use **gc\_GetCallInfo()**

**gc\_GetLinedevState()**

retrieves the status of the line device

**gc\_GetVer()**

retrieves the version number of a specified software component

**gc\_ReqMoreInfo()**

requests more information such as ANI or DNIS

**gc\_SendMoreInfo()**

sends more information to the remote side

**gc\_SetBilling()**

sets billing information

**gc\_SetCallingNum()**

deprecated: use **gc\_SetConfigData()** if supported by the technology

**gc\_SetChanState()**

sets the channel state of the indicated channel

## 1.4 Advanced Call Model Functions

The following functions are used to place calls on hold, retrieve held calls, and transfer calls. See the *Global Call API Programming Guide* for more information about the advanced call state model.

**gc\_BlindTransfer()**

initiates and completes a one-step (unsupervised) transfer

**gc\_CompleteTransfer()**

completes the transfer of a call

**gc\_HoldCall()**

places an active call on hold

**gc\_RetrieveCall()**

retrieves a call from the OnHold state

**gc\_SetUpTransfer()**

initiates a supervised call transfer

**gc\_SwapHold()**

switches between an active call and a call on hold or pending transfer

## 1.5 System Controls and Tools Functions

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

**gc\_Close()**

closes a previously opened device

**gc\_CRN2LineDev()**

maps a CRN to its line device ID

**gc\_ErrorInfo()**

provides error information about a failed function

**gc\_ErrorValue()**  
 deprecated: use **gc\_ErrorInfo()**

**gc\_GetCallProgressParm()**  
 retrieves protocol call progress parameters

**gc\_GetCallState()**  
 retrieves the state of a call

**gc\_GetCRN()**  
 retrieves a call reference number

**gc\_GetCTInfo()**  
 retrieves CT Bus time slot information

**gc\_GetLineDev()**  
 retrieves a line device associated with an event

**gc\_GetMetaEvent()**  
 retrieves the metaevent structure for the current SRL event

**gc\_GetMetaEventEx()**  
 retrieves the metaevent structure for the current SRL eventmaps (Windows only)

**gc\_GetNetworkH()**  
 deprecated: use **gc\_GetResourceH()** if supported by the technology

**gc\_GetParm()**  
 retrieves the value of the specified parameter

**gc\_GetResourceH()**  
 retrieves a device (network, voice, or media) handle

**gc\_GetUsrAttr()**  
 retrieves the user-defined attribute

**gc\_GetXmitSlot()**  
 retrieves the network CT Bus time slot number

**gc\_Listen()**  
 connects a channel to a network CT Bus time slot

**gc\_Open()**  
 deprecated: use **gc\_OpenEx()**

**gc\_OpenEx()**  
 opens a Global Call device and sets a user defined attribute

**gc\_ResetLineDev()**  
 resets the line device state and disconnects calls

**gc\_ResultInfo()**  
 retrieves information about solicited and unsolicited events

**gc\_ResultMsg()**  
 deprecated: use **gc\_ResultInfo()**

**gc\_ResultValue()**  
 deprecated: use **gc\_ResultInfo()**

**gc\_SetCallProgressParm()**

overrides protocol default call progress parameters

**gc\_SetConfigData()**

updates the configuration data

**gc\_SetEvtMsk()**

deprecated: use **gc\_SetConfigData()** if supported by the technology

**gc\_SetParm()**

sets the default parameters

**gc\_SetUsrAttr()**

sets an attribute defined by the user

**gc\_Start()**

starts and initializes all call control libraries

**gc\_Stop()**

stops call control libraries and releases resources

**gc\_UnListen()**

disconnects a channel from the network CT Bus time slot

## 1.6 Voice and Media Functions

The following functions support the association of voice and media resources with a Global Call line device.

**gc\_Attach()**

deprecated: use **gc\_AttachResource()** if supported by the technology

**gc\_AttachResource()**

attaches a media resource to the specified line device and provides optional capability exchange

**gc\_Detach()**

detaches a voice or media resource from a line device

**gc\_GetResourceH()**

retrieves the network, voice, or media device handle

**gc\_GetVoiceH()**

deprecated: use **gc\_GetResourceH()** if supported by the technology

**gc\_LoadDxParm()**

sets voice parameters associated with a line device

## 1.7 ISDN Interface Specific Functions

The following functions support ISDN-specific functionality.

**gc\_CallProgress()**

provides information about the progress of a call

**gc\_GetCallInfo()**  
retrieves information associated with a call

**gc\_GetFrame()**  
deprecated: use **gc\_Extension()**

**gc\_GetInfoElem()**  
deprecated: use **gc\_GetUserInfo()**

**gc\_GetNetCRV()**  
deprecated: use **gc\_Extension()**

**gc\_GetSigInfo()**  
retrieves the signaling information of an incoming message

**gc\_GetUserInfo()**  
retrieves technology-specific user information

**gc\_HoldACK()**  
accepts a hold request from remote equipment

**gc\_HoldRej()**  
rejects a hold request from remote equipment

**gc\_ReqANI()**  
requests the remote side to return ANI

**gc\_RetrieveAck()**  
accepts a retrieve request from remote equipment

**gc\_RetrieveRej()**  
rejects a retrieve request from remote equipment

**gc\_SetInfoElem()**  
deprecated: use **gc\_SetUserInfo()**

**gc\_SndFrame()**  
deprecated: use **gc\_Extension()**

**gc\_SndMsg()**  
deprecated: use **gc\_Extension()**

**gc\_StartTrace()**  
starts the logging of debug information

**gc\_StopTrace()**  
stops the logging of debug information

## 1.8 Global Call Alarm Management System (GCAMS) Functions

The following functions are used to configure and manage the Global Call Alarm Management System (GCAMS).

**gc\_AlarmName()**  
returns the name of the alarm for the current alarm event



**gc\_AlarmNumber()**  
returns the alarm number for the current alarm event

**gc\_AlarmNumberToName()**  
converts an alarm number to a name

**gc\_AlarmSourceObjectID()**  
retrieves ASO ID for the current alarm event

**gc\_AlarmSourceObjectIDToName()**  
converts the ASO ID to the ASO name

**gc\_AlarmSourceObjectName()**  
returns the ASO name for the current alarm event

**gc\_AlarmSourceObjectNameToID()**  
converts the ASO name to the ASO ID

**gc\_GetAlarmConfiguration()**  
retrieves alarm configuration parameter values

**gc\_GetAlarmFlow()**  
retrieves a value indicating which alarms are sent to the application

**gc\_GetAlarmParm()**  
retrieves the parameter data associated with the parameter set ID

**gc\_GetAlarmSourceObjectList()**  
retrieves a list of all alarm source objects

**gc\_GetAlarmSourceObjectNetworkID()**  
retrieves the ID of the layer 1 ASO

**gc\_SetAlarmConfiguration()**  
sets alarm configuration parameter values

**gc\_SetAlarmFlow()**  
configures which alarms are sent to the application

**gc\_SetAlarmNotifyAll()**  
sets the notification attribute of all alarms originating from an ASO

**gc\_SetAlarmParm()**  
sets the data associated with the alarm parameter

**gc\_StopTransmitAlarms()**  
stops the transmission of one or more alarms

**gc\_TransmitAlarms()**  
starts the transmission of alarms

## 1.9 Feature Transparency and Extension (FTE) Functions

The following functions are used to extend the generic Global Call API to access technology-specific or protocol-specific features that are unique to a given network interface.

### **gc\_Extension()**

provides a generic API interface to support technology-specific features

### **gc\_GetUserInfo()**

retrieves technology-specific user information

### **gc\_SetUserInfo()**

permits the specification of technology-specific user information

## 1.10 Real Time Configuration Management (RTCM) Functions

The following functions are used to query, retrieve, or update configuration parameter data dynamically.

### **gc\_GetConfigData()**

retrieves configured parameter values for a given target object

### **gc\_QueryConfigData()**

queries the configuration data

### **gc\_SetConfigData()**

updates the configuration data

## 1.11 Global Call Service Request (GCSR) Functions

The following functions are used to manage the sending of a request to a remote device that provides some kind of service, and handle the response.

### **gc\_ReqService()**

requests a service from a remote device

### **gc\_RespService()**

returns a response to a requested service

## 1.12 GC\_PARM\_BLK Utility Functions

The following functions are used to create and manage [GC\\_PARM\\_BLK](#) structures required by specific Global Call features.

### **gc\_util\_delete\_parm\_blk()**

deletes the specified GC\_PARM\_BLK

**gc\_util\_find\_parm()**

finds a parameter in a GC\_PARM\_BLK

**gc\_util\_insert\_parm\_ref()**

adds a parameter by reference to a GC\_PARM\_BLK

**gc\_util\_insert\_parm\_val()**

adds a parameter by value to a GC\_PARM\_BLK

**gc\_util\_next\_parm()**

finds the next parameter in a GC\_PARM\_BLK

## 1.13 Deprecated Functions

The following are functions for which there is now a preferred alternative. Deprecated functions are still supported but may eventually be phased out over time. This list gives the names of the deprecated functions and the preferred equivalent functions.

**gc\_Attach()**

use **gc\_AttachResource()**

**gc\_CCLibStatus()**

use **gc\_CCLibStatusEx()**

**gc\_CCLibStatusAll()**

use **gc\_CCLibStatusEx()**

**gc\_ErrorValue()**

use **gc\_ErrorInfo()**

**gc\_GetANI()**

use **gc\_GetCallInfo()**

**gc\_GetDNIS()**

use **gc\_GetCallInfo()**

**gc\_GetFrame()**

use **gc\_Extension()**

**gc\_GetInfoElem()**

use **gc\_GetUserInfo()**

**gc\_GetNetCRV()**

use **gc\_Extension()**

**gc\_GetNetworkH()†**

use **gc\_GetResourceH()**

**gc\_GetVoiceH()†**

use **gc\_GetResourceH()**

**gc\_Open()**

use **gc\_OpenEx()**

**gc\_ReleaseCall()**

use **gc\_ReleaseCallEx()**

**gc\_ResultMsg()**  
     use **gc\_ResultInfo()**  
  
**gc\_ResultValue()**  
     use **gc\_ResultInfo()**  
  
**gc\_SetCallingNum()**†  
     use **gc\_SetConfigData()**  
  
**gc\_SetEvtMsk()**†  
     use **gc\_SetConfigData()**  
  
**gc\_SetInfoElem()**  
     use **gc\_SetUserInfo()**  
  
**gc\_SndFrame()**  
     use **gc\_Extension()**  
  
**gc\_SndMsg()**  
     use **gc\_Extension()**

**Note:** The dagger (†) next to a function name indicates that the function is not deprecated for all call control libraries. See the function description page for more information.

## 1.14 Global Call Function Support by Technology

Table 1 provides an alphabetical listing of all the Global Call API functions. The table indicates which technologies are supported for each of the functions and whether the function is supported as described in this *Global Call API Library Reference* or whether you need to refer to the appropriate *Global Call Technology User's Guide* for additional information.

**Table 1. Global Call Function Support by Technology**

Function	Technology								
	T-1/E-1 (Springware)	ISDN (Springware)	Analog (Springware)	T-1/E-1 (DM3)	ISDN (DM3)	Analog (DM3)	SS7	IP (Host- Based Stack)	IP (Embedded Stack)
<b>gc_AcceptCall()</b>	S*	S*	S*	S*	S*	S*	S*	S*	S*
<b>gc_AlarmName()</b>	S	S	NS	S	S	NS	NS	S	NS
<b>gc_AlarmNumber()</b>	S	S	NS	S	S	NS	NS	S	NS
<b>gc_AlarmNumberToName()</b>	S	S	NS	S	S	NS	NS	S	NS
<p> <b>B</b> = ISDN BRI only  <b>D</b> = ISDN PRI, DPNSS protocols only  <b>NS</b> = Not supported  <b>P</b> = PDKRT only  <b>PR</b> = ISDN PRI only  <b>Q</b> = ISDN PRI, Q.SIG protocol only  <b>S</b> = Supported  † = Not supported when using PDK analog  * = Variances; refer to the appropriate <i>Global Call Technology User's Guide</i>. </p>									

Table 1. Global Call Function Support by Technology (Continued)

Function	Technology								
	T-1/E-1 (Springware)	ISDN (Springware)	Analog (Springware)	T-1/E-1 (DM3)	ISDN (DM3)	Analog (DM3)	SS7	IP (Host- Based Stack)	IP (Embedded Stack)
<a href="#">gc_AlarmSourceObjectID( )</a>	S	S	NS	S	S	NS	NS	S	NS
<a href="#">gc_AlarmSourceObjectIDToName( )</a>	S	S	NS	S	S	NS	NS	S	NS
<a href="#">gc_AlarmSourceObjectName( )</a>	S	S	NS	S	S	NS	NS	S	NS
<a href="#">gc_AlarmSourceObjectNameToID( )</a>	S	S	NS	S	S	NS	NS	S	NS
<a href="#">gc_AnswerCall( )</a>	S*	S*	S*	S*	S*	S*	S*	S*	S*
<a href="#">gc_Attach( )</a> (deprecated)	S	NS	NS	S	S	S	S	NS	NS
<a href="#">gc_AttachResource( )</a>	S	NS	NS	S	S	S	NS	S	S*
<a href="#">gc_BlindTransfer( )</a>	P*	NS	NS	S*	NS	NS	NS	NS	NS
<a href="#">gc_CallAck( )</a>	S*	S*	NS	NS	S*	NS	S*	S*	NS
<a href="#">gc_CallProgress( )</a>	NS	S*	NS	NS	NS	NS	NS	NS	NS
<a href="#">gc_CCLibIDToName( )</a>	S	S	S	S	S	S	S	S	S
<a href="#">gc_CCLibNameToID( )</a>	S	S	S	S	S	S	S	S	S
<a href="#">gc_CCLibStatus( )</a> (deprecated)	S	S	S	S	S	S	S	S	S
<a href="#">gc_CCLibStatusAll( )</a> (deprecated)	S	S	S	S	S	S	S	S	S
<a href="#">gc_CCLibStatusEx( )</a>	S	S	S	S	S	S	S	S	S
<a href="#">gc_Close( )</a>	S*	S	S	S*	S	S	S	S	S
<a href="#">gc_CompleteTransfer( )</a>	P*	NS	NS	NS	NS	NS	NS	NS	NS
<a href="#">gc_CRN2LineDev( )</a>	S	S	S	S	S	S	S	S	S
<a href="#">gc_Detach( )</a>	S*	NS	NS	S*	S	S	S	S	S*
<a href="#">gc_DropCall( )</a>	S*	S*	S*	S*	S*	S*	S*	S*	S*
<a href="#">gc_ErrorInfo( )</a>	S	S	S	S	S	S	S	S	S
<a href="#">gc_ErrorValue( )</a> (deprecated)	S	S	S	S	S	S	S*	S	S
<a href="#">gc_Extension( )</a>	P*	S*	P	NS	S*	NS	S*	S*	S*
<a href="#">gc_GetAlarmConfiguration( )</a>	S	S	NS	S	S	NS	NS	S	NS
<a href="#">gc_GetAlarmFlow( )</a>	S	S	NS	S	S	NS	NS	S	NS
<a href="#">gc_GetAlarmParm( )</a>	S	S	NS	NS	NS	NS	NS	S*	S*
<p> <b>B</b> = ISDN BRI only  <b>D</b> = ISDN PRI, DPNSS protocols only  <b>NS</b> = Not supported  <b>P</b> = PDKRT only  <b>PR</b> = ISDN PRI only  <b>Q</b> = ISDN PRI, Q.SIG protocol only  <b>S</b> = Supported  <b>†</b> = Not supported when using PDK analog  <b>*</b> = Variances; refer to the appropriate <i>Global Call Technology User's Guide</i>. </p>									

Table 1. Global Call Function Support by Technology (Continued)

Function	Technology								
	T-1/E-1 (Springware)	ISDN (Springware)	Analog (Springware)	T-1/E-1 (DM3)	ISDN (DM3)	Analog (DM3)	SS7	IP (Host- Based Stack)	IP (Embedded Stack)
<a href="#">gc_GetAlarmSourceObjectList( )</a>	S	S	NS	S	S	NS	NS	S	NS
<a href="#">gc_GetAlarmSourceObjectNetworkID( )</a>	S	S	NS	S	S	NS	NS	S	NS
<a href="#">gc_GetANI( )</a> (deprecated)	S	S*	S*	S	S*	S*	S	NS	S*
<a href="#">gc_GetBilling( )</a>	NS	S*	NS	NS	NS	NS	NS	NS	NS
<a href="#">gc_GetCallInfo( )</a>	S*	S*	S*	S*	S*	S*	S*	S*	S*
<a href="#">gc_GetCallProgressParm( )</a>	P	NS	NS	NS	NS	NS	NS	NS	NS
<a href="#">gc_GetCallState( )</a>	S	S	S	S	S	S	S	S	S
<a href="#">gc_GetConfigData( )</a>	P	S*	P	NS	NS	NS	NS	NS	NS
<a href="#">gc_GetCRN( )</a>	S	S	S	S	S	S	S	S	S
<a href="#">gc_GetCTInfo( )</a>	P	NS	NS	S	S	S	NS	NS	S
<a href="#">gc_GetDNIS( )</a> (deprecated)	S	S*	NS	S	S*	S	S*	NS	S*
<a href="#">gc_GetFrame( )</a> (deprecated)	NS	S	NS	NS	S	NS	NS	NS	NS
<a href="#">gc_GetInfoElem( )</a> (deprecated)	NS	S	NS	NS	NS	NS	NS	NS	NS
<a href="#">gc_GetLineDev( )</a>	S	S	S	S	S	S	S	S	S
<a href="#">gc_GetLinedevState( )</a>	NS	S	NS	S	S	S	S	NS	S
<a href="#">gc_GetMetaEvent( )</a>	S	S	S	S	S	S	S	S	S
<a href="#">gc_GetMetaEventEx( )</a> (Windows extended asynchronous model only)	S	S	S	S	S	S	S	S	S
<a href="#">gc_GetNetCRV( )</a> (deprecated)	NS	S	NS	NS	S*	NS	NS	NS	NS
<a href="#">gc_GetNetworkH( )</a> (deprecated)	S	S	NS	S	S	S	S*	NS	S
<a href="#">gc_GetParm( )</a>	S*	S*	S*	S*	S*	S*	S*	NS	NS
<a href="#">gc_GetResourceH( )</a>	S	S	S	S	S	S	NS	S*	S*
<a href="#">gc_GetSigInfo( )</a>	NS	S*	NS	NS	S*	NS	S*	NS	NS
<a href="#">gc_GetUserInfo( )</a>	NS	S*	NS	NS	NS	NS	NS	NS	NS
<a href="#">gc_GetUsrAttr( )</a>	S	S	S	S	S	S	S	S	S
<a href="#">gc_GetVer( )</a>	S	S	S	NS	NS	NS	S	S	S
<b>B</b> = ISDN BRI only <b>D</b> = ISDN PRI, DPNSS protocols only <b>NS</b> = Not supported <b>P</b> = PDKRT only <b>PR</b> = ISDN PRI only <b>Q</b> = ISDN PRI, Q.SIG protocol only <b>S</b> = Supported <b>†</b> = Not supported when using PDK analog <b>*</b> = Variances; refer to the appropriate <i>Global Call Technology User's Guide</i> .									

Table 1. Global Call Function Support by Technology (Continued)

Function	Technology								
	T-1/E-1 (Springware)	ISDN (Springware)	Analog (Springware)	T-1/E-1 (DM3)	ISDN (DM3)	Analog (DM3)	SS7	IP (Host- Based Stack)	IP (Embedded Stack)
<a href="#">gc_GetVoiceH( )</a> (deprecated)	S	S	S	S	S	S	S	NS	S
<a href="#">gc_GetXmitSlot( )</a>	P	NS	NS	S	S	S	S	S*	S*
<a href="#">gc_HoldACK( )</a>	NS	B,D,Q	NS	NS	NS	NS	NS	NS	NS
<a href="#">gc_HoldCall( )</a>	P*	B,D,Q*	NS	NS	NS	NS	S*	NS	NS
<a href="#">gc_HoldRej( )</a>	NS	B,D,Q	NS	NS	NS	NS	NS	NS	NS
<a href="#">gc_LinedevToCCLIBID( )</a>	S	S	S	S	S	S	S	S	S
<a href="#">gc_Listen( )</a>	P	NS	NS	S	S	S	S	S*	S*
<a href="#">gc_LoadDxParm( )</a>	P	NS	S	NS	NS	NS	NS	NS	NS
<a href="#">gc_MakeCall( )</a>	S*	S*	S*	S*	S*	S*	S*	S*	S*
<a href="#">gc_Open( )</a> (deprecated)	S	S	S	S	S	S	S	NS	S
<a href="#">gc_OpenEx( )</a>	S*	S*	S*	S*	S*	S*	S*	S*	S*
<a href="#">gc_QueryConfigData( )</a>	P*	S	NS	NS	NS	NS	NS	NS	NS
<a href="#">gc_ReleaseCall( )</a> (deprecated)	S	S	S	S	S	S	S	NS	S
<a href="#">gc_ReleaseCallEx( )</a>	S	S*	P*	S	S*	S*	S	S*	S
<a href="#">gc_ReqANI( )</a>	NS	S*	NS	NS	NS	NS	NS	NS	NS
<a href="#">gc_ReqMoreInfo( )</a>	P	S	NS	NS	S*	NS	S	NS	NS
<a href="#">gc_ReqService( )</a>	NS	NS	NS	NS	NS	NS	NS	S*	NS
<a href="#">gc_ResetLineDev( )</a>	S*	S*	S*	S	S*	S*	S*	S	S
<a href="#">gc_RespService( )</a>	NS	B*	NS	NS	NS	NS	NS	S*	NS
<a href="#">gc_ResultInfo( )</a>	S	S	S	S	S	S	S	S	S
<a href="#">gc_ResultMsg( )</a> (deprecated)	S	S	S	S	S	S	S	NS	S
<a href="#">gc_ResultValue( )</a> (deprecated)	S	S	S	S	S	S	S*	NS	S
<a href="#">gc_RetrieveAck( )</a>	NS	B,D,Q	NS	NS	NS	NS	NS	NS	NS
<a href="#">gc_RetrieveCall( )</a>	P*	B,D,Q	NS	NS	NS	NS	S*	NS	NS
<a href="#">gc_RetrieveRej( )</a>	NS	B,D,Q	NS	NS	NS	NS	NS	NS	NS
<a href="#">gc_SendMoreInfo( )</a>	P	S	NS	NS	S*	NS	S	NS	NS
<p> <b>B</b> = ISDN BRI only  <b>D</b> = ISDN PRI, DPNSS protocols only  <b>NS</b> = Not supported  <b>P</b> = PDKRT only  <b>PR</b> = ISDN PRI only  <b>Q</b> = ISDN PRI, Q.SIG protocol only  <b>S</b> = Supported  <b>†</b> = Not supported when using PDK analog  <b>*</b> = Variances; refer to the appropriate <i>Global Call Technology User's Guide</i>. </p>									

Table 1. Global Call Function Support by Technology (Continued)

Function	Technology								
	T-1/E-1 (Springware)	ISDN (Springware)	Analog (Springware)	T-1/E-1 (DM3)	ISDN (DM3)	Analog (DM3)	SS7	IP (Host- Based Stack)	IP (Embedded Stack)
<a href="#">gc_SetAlarmConfiguration( )</a>	S	S	NS	S	S	NS	NS	S	NS
<a href="#">gc_SetAlarmFlow( )</a>	S	S	NS	S	S	NS	NS	S	NS
<a href="#">gc_SetAlarmNotifyAll( )</a>	S	S	NS	S	S	NS	NS	S	NS
<a href="#">gc_SetAlarmParm( )</a>	S	S	NS	NS	NS	NS	NS	S*	S*
<a href="#">gc_SetBilling( )</a>	S*	S*	NS	NS	NS	NS	S*	NS	NS
<a href="#">gc_SetCallingNum( )</a> (deprecated)	S	S*	NS	S	S*	S	S	NS	S
<a href="#">gc_SetCallProgressParm( )</a>	P	NS	NS	NS	NS	NS	NS	NS	NS
<a href="#">gc_SetChanState( )</a>	S*	PR*	NS	S*	S*	S	S*	NS	NS
<a href="#">gc_SetConfigData( )</a>	P*	S*	P	NS	NS	NS	S*	S*	S*
<a href="#">gc_SetEvtMsk( )</a> (deprecated)	S*	S*	S	S	S*	S	S	NS	S
<a href="#">gc_SetInfoElem( )</a> (deprecated)	NS	S*	NS	NS	NS	NS	S*	NS	NS
<a href="#">gc_SetParm( )</a>	S*	S*	S†	S*	S*	S*	S*	NS	NS
<a href="#">gc_SetUpTransfer( )</a>	P*	NS	NS	NS	NS	NS	NS	NS	NS
<a href="#">gc_SetUserInfo( )</a>	NS	S*	NS	NS	S*	NS	NS	S*	S*
<a href="#">gc_SetUsrAttr( )</a>	S	S	S	S	S	S	S	S	S
<a href="#">gc_SndFrame( )</a> (deprecated)	NS	S*	NS	NS	S*	NS	NS	NS	NS
<a href="#">gc_SndMsg( )</a> (deprecated)	NS	S*	NS	NS	S*	NS	S*	NS	NS
<a href="#">gc_Start( )</a>	S*	S	S*	S	S	S	S	S*	S
<a href="#">gc_StartTrace( )</a>	P*	S*	P*	NS	S*	NS	S*	NS	NS
<a href="#">gc_Stop( )</a>	S*	S	S	S	S	S	S	S	S
<a href="#">gc_StopTrace( )</a>	P	S*	P	NS	S*	NS	S*	NS	NS
<a href="#">gc_StopTransmitAlarms( )</a>	S	S	NS	S	S	NS	NS	NS	NS
<a href="#">gc_SwapHold( )</a>	P*	NS	NS	NS	NS	NS	NS	NS	NS
<a href="#">gc_TransmitAlarms( )</a>	S	S	NS	S	S	NS	NS	NS	NS
<a href="#">gc_UnListen( )</a>	P	NS	NS	S	S	S	S	S*	S*
<a href="#">gc_util_delete_parm_blk( )</a>	S	S	S	S	S	S	S	S	S
<b>B</b> = ISDN BRI only <b>D</b> = ISDN PRI, DPNSS protocols only <b>NS</b> = Not supported <b>P</b> = PDKRT only <b>PR</b> = ISDN PRI only <b>Q</b> = ISDN PRI, Q.SIG protocol only <b>S</b> = Supported <b>†</b> = Not supported when using PDK analog <b>*</b> = Variances; refer to the appropriate <i>Global Call Technology User's Guide</i> .									



Table 1. Global Call Function Support by Technology (Continued)

Function	Technology								
	T-1/E-1 (Springware)	ISDN (Springware)	Analog (Springware)	T-1/E-1 (DM3)	ISDN (DM3)	Analog (DM3)	SS7	IP (Host- Based Stack)	IP (Embedded Stack)
<code>gc_util_find_parm( )</code>	S	S	S	S	S	S	S	S	S
<code>gc_util_insert_parm_ref( )</code>	S	S	S	S	S	S	S	S	S
<code>gc_util_insert_parm_val( )</code>	S	S	S	S	S	S	S	S	S
<code>gc_util_next_parm( )</code>	S	S	S	S	S	S	S	S	S
<code>gc_WaitCall( )</code>	S	S*	S	S	S*	S	S	S	S
<p> <b>B</b> = ISDN BRI only  <b>D</b> = ISDN PRI, DPNSS protocols only  <b>NS</b> = Not supported  <b>P</b> = PDKRT only  <b>PR</b> = ISDN PRI only  <b>Q</b> = ISDN PRI, Q.SIG protocol only  <b>S</b> = Supported  <b>†</b> = Not supported when using PDK analog  <b>*</b> = Variances; refer to the appropriate <i>Global Call Technology User's Guide</i>. </p>									



This chapter provides an alphabetical reference to the functions in the Global Call API library. A general description of the function syntax convention is provided before the detailed function information.

- Notes:**
1. Unless otherwise indicated, the functions described in this chapter are available for application development on all supported platforms and technologies. See the “Platform and Technology” line in the function header table of each function for specific platform and technology applicability.
  2. All function prototypes are in the *gclib.h* header file.

## 2.1 Function Syntax Conventions

The Global Call API functions use the following format:

```
int gc_function(reference, param1, param2, ..., paramN, mode)
```

where:

**int**

represents an integer return value that indicates if the function succeeded or failed. Possible values are:

- 0 if the function succeeds
- <0 if the function fails

**gc\_function**

represents the function name

**reference**

represents an input field that directs the function to a specific line device or call. Possible data types for this parameter are:

- LINEDEV for a line device
- CRN for a call

**param1**

represents the first parameter

**paramN**

represents the last parameter

**mode**

indicates the mode of execution. Possible values are:

- EV\_ASYNC for asynchronous mode execution
- EV\_SYNC for synchronous mode execution

**Note:** In the C language coding example listed in the Example section for each function, the symbolic constant GC\_SUCCESS is used as the function return value. GC\_SUCCESS is defined in the *gcerr.h* header file to equate to 0.

## **gc\_AcceptCall()**

**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 call handling

**Mode:** asynchronous or synchronous

**Platform and** All†

**Technology:** †See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

The **gc\_AcceptCall()** function indicates to the originator that the call will be answered, but additional work needs to be done. The function provides a response to a destination party request (GCEV\_OFFERED event or termination of the **gc\_WaitCall()** function) that acknowledges that the call has been received but is not yet answered (for example, the phone is ringing). Upon successful completion of the **gc\_AcceptCall()** function, the call state changes from the Offered state to the Accepted state.

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 and insert application in which the outbound dialing process may be time consuming.

Parameter	Description
<b>crn</b>	call reference number
<b>rings</b>	specifies how long (the number of rings) the protocol handler will wait before notifying the calling entity. Valid values are: <ul style="list-style-type: none"><li>• 0 to 14 – the call is accepted after the specified number of rings</li><li>• 15 or greater – the call will ring forever; this allows the application to accept calls without answering them. However, the call can still be answered after 15 or more rings using <b>gc_AnswerCall()</b>.</li></ul> For protocols not using rings, the <b>rings</b> parameter is ignored.
<b>mode</b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

## ■ Termination Events

### GCEV\_ACCEPT

indicates successful completion of the **gc\_AcceptCall()** function, that is, the inbound call was accepted

### GCEV\_TASKFAIL

indicates that the function failed. For more information, see the “Error Handling” section in the *Global Call API Programming Guide*.

## ■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value **EGC\_UNSUPPORTED** will be the Global Call value returned when the **gc\_ErrorInfo()** function is used to retrieve the error code.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function for error information. If the **GCEV\_TASKFAIL** event is received, use the **gc\_ResultInfo()** function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
#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)
 * 4. The event is determined to be a GCEV_OFFERED event
 */

int accept_call(void)
{
    CRN          crn;          /* Call Reference Number */
    GC_INFO      gc_error_info; /* GlobalCall error information data */
    /*
     * Accept the incoming call.
     */
    crn = metaevent.crn;
    if (gc_AcceptCall(crn, 0, EV_ASYNC) != GC_SUCCESS) {
        /* process error return as shown */

        gc_ErrorInfo( &gc_error_info );
        printf ("Error gc_AcceptCall() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
        CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
            metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
            gc_error_info.ccLibId, gc_error_info.ccLibName,
            gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
}
```

```
    }  
    /*  
    * gc_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);  
}
```

■ **See Also**

- [gc\\_WaitCall\(\)](#)
- [gc\\_AnswerCall\(\)](#)

## gc\_AlarmName( )

**Name:** int gc\_AlarmName(metaeventp, alarm\_name)

**Inputs:** METAEVENT \*metaeventp    • pointer to metaevent  
char \*\*alarm\_name                    • pointer to address of pointer to alarm name

**Returns:** 0 if successful  
          <0 if failure

**Includes:** gclib.h  
            gcerr.h

**Category:** GCAMS

**Mode:** synchronous

**Platform and** Springware: T-1/E-1, ISDN

**Technology:** DM3: T-1/E-1, ISDN  
                IP (host-based stack)

---

### ■ Description

The **gc\_AlarmName()** function returns the name of the alarm for the current alarm event. The function is used after a GCEV\_ALARM event is received to retrieve the name of the alarm. The information retrieved by the function can be used in reports or for screen display. See the appropriate *Global Call Technology User's Guide* for a list of possible alarm names.

Parameter	Description
metaeventp	points to the metaevent
alarm_name	points to the destination for the pointer to the alarm name. The destination cannot be NULL.

### ■ Termination Events

None

### ■ Cautions

- Do not overwrite the space pointed to by **alarm\_name** as this is private, internal space used by GCAMS.
- The **gc\_AlarmName()** function can only be called for GCEV\_ALARM events.
- The data that is to be retrieved by the **gc\_AlarmName()** function is valid only until the next event is requested.

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*.

All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

### ■ Example

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

    METAEVENT    metaevent;
    char          *alarm_name;
    int           rc;

/*
-- This code assumes that the current event is GCEV_ALARM
-- event and that gc_GetMetaEvent() has already been called
-- to place the current event information into metaevent
*/

    rc = gc_AlarmName(&metaevent, &alarm_name);
    if (rc < 0)
    {
        /* get and process the error */
    }
    else
    {
        printf("Alarm name is %s\n", alarm_name);
    }
```

### ■ See Also

- [gc\\_AlarmNumber\(\)](#)



## gc\_AlarmNumber( )

**Name:** int gc\_AlarmNumber(metaeventp, alarm\_number)

**Inputs:** METAEVENT \*metaeventp • pointer to metaevent  
long \*alarm\_number • pointer to location for storing the alarm number

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** GCAMS

**Mode:** synchronous

**Platform and** Springware: T-1/E-1, ISDN

**Technology:** DM3: T-1/E-1, ISDN  
IP (host-based stack)

### ■ Description

The **gc\_AlarmNumber( )** function returns the alarm number for the current alarm event. The function is used to retrieve the alarm number after a GCEV\_ALARM event is received. The information retrieved by the function can be used in reports or for screen display.

Parameter	Description
<b>metaeventp</b>	points to the metaevent
<b>alarm_number</b>	points to the destination for the alarm number. The destination cannot be NULL.

### ■ Termination Events

None

### ■ Cautions

- The **gc\_AlarmNumber( )** function can only be called for GCEV\_ALARM events.
- The data that is to be retrieved by the **gc\_AlarmNumber( )** function is valid only until the next event is requested.

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

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

    METAEVENT    metaevent;
    long         alarm_number;
    int          rc;

/*
-- This code assumes that the current event is GCEV_ALARM
-- event and that gc_GetMetaEvent() has already been called
-- to place the current event information into metaevent
*/

    rc = gc_AlarmNumber(&metaevent, &alarm_number);
    if (rc < 0)
    {
        /* get and process the error */
    }
    else
    {
        printf("Alarm number is %d\n", alarm_number);
    }
```

## ■ See Also

- [gc\\_AlarmName\( \)](#)
- [gc\\_AlarmNumberToName\( \)](#)

## `gc_AlarmNumberToName()`

**Name:** `int gc_AlarmNumberToName(aso_id, alarm_number, name)`

**Inputs:**

<code>unsigned long aso_id</code>	• alarm source object ID
<code>long alarm_number</code>	• alarm number
<code>char **name</code>	• pointer to address where pointer to name is to be returned

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** GCAMS

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN  
DM3: T-1/E-1, ISDN  
IP (host-based stack)

---

### ■ Description

The `gc_AlarmNumberToName()` function converts an alarm number to a name for a specified alarm source object. This function can be used to determine the alarm name for a given alarm number.

Parameter	Description
<b>aso_id</b>	alarm source object (ASO) ID
<b>alarm_number</b>	alarm number to be converted
<b>name</b>	points to the destination for the pointer to the alarm name. The destination cannot be NULL.

### ■ Termination Events

None

### ■ Cautions

Do not overwrite the space pointed to by **alarm\_name** as it is private, internal space used by GCAMS.

### ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology

specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

### ■ Example

```
#include <stdio.h>
#include <stdlib.h>
#include <gclib.h>
#include <gcerr.h>
#include <dtilib.h>    /* for ASO symbols */

char      *alarm_name;
int        rc;

rc = gc_AlarmNumberToName(ALARM_SOURCE_ID_SPRINGWARE_E1, DTE1_LOS, &alarm_name);
if (rc < 0)
{
    /* get and process the error */
}
else
{
    printf("Alarm name for DTE1_LOS is %s\n", alarm_name);
}
```

### ■ See Also

- [\*\*\*gc\\_AlarmNumber\(\)\*\*\*](#)

## gc\_AlarmSourceObjectID( )

**Name:** int gc\_AlarmSourceObjectID(metaeventp, aso\_id)

**Inputs:** METAEVENT \*metaeventp • pointer to metaevent  
 unsigned long \*aso\_id • pointer to where to store alarm source object ID

**Returns:** 0 if successful  
 <0 if failure

**Includes:** gclib.h  
 gcerr.h

**Category:** GCAMS

**Mode:** synchronous

**Platform and** Springware: T-1/E-1, ISDN

**Technology:** DM3: T-1/E-1, ISDN  
 IP (host-based stack)

### ■ Description

The **gc\_AlarmSourceObjectID( )** function returns the alarm source object (ASO) ID for the current alarm event. The function is used to retrieve the ASO ID after a GCEV\_ALARM event is received. The information retrieved by the function can be used in reports or for screen display.

Parameter	Description
<b>metaeventp</b>	points to metaevent
<b>aso_id</b>	points to the destination for the alarm source object ID. The destination cannot be NULL.

### ■ Termination Events

None

### ■ Cautions

- The **gc\_AlarmSourceObjectID( )** function can only be called for GCEV\_ALARM events.
- The data that is to be retrieved by the **gc\_AlarmSourceObjectID( )** function is valid only until the next event is requested.

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

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

    METAEVENT      metaevent;
    unsigned long   aso_id;
    int             rc;

/*
-- This code assumes that the current event is GCEV_ALARM
-- event and that gc_GetMetaEvent() has already been called
-- to place the current event information into metaevent
*/

    rc = gc_AlarmSourceObjectID(&metaevent, &aso_id);
    if (rc < 0)
    {
        /* get and process the error */
    }
    else
    {
        printf("Alarm source object ID is %d\n", aso_id);
    }
```

## ■ See Also

None

## `gc_AlarmSourceObjectIDToName()`

**Name:** `int gc_AlarmSourceObjectIDToName(aso_id, aso_name)`

**Inputs:** `unsigned long aso_id` • destination for alarm source object ID  
`char **aso_name` • pointer to address of alarm source object name

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** GCAMS

**Mode:** synchronous

**Platform and** Springware: T-1/E-1, ISDN

**Technology:** DM3: T-1/E-1, ISDN  
IP (host-based stack)

### ■ Description

The `gc_AlarmSourceObjectIDToName()` function converts the alarm source object (ASO) ID to the ASO name. The function is used after a GCEV\_ALARM event is received to convert the alarm source object ID to the name of the alarm source object. The information retrieved by the function can be used in reports or for screen display.

Parameter	Description
<b>aso_id</b>	alarm source object ID
<b>aso_name</b>	points to the location where the pointer to the alarm source object name is to be stored. The <b>aso_name</b> cannot be NULL.

### ■ Termination Events

None

### ■ Cautions

Do not overwrite the value that is returned in **aso\_name** as this points to private, internal GCAMS space.

### ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

### ■ Example

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

char      *aso_name;
int       rc;

rc = gc_AlarmSourceObjectIDToName(ALARM_SOURCE_ID_SPRINGWARE_E1, &aso_name);
if (rc < 0)
{
    /* get and process the error */
}
else
{
    printf("ALARM_SOURCE_ID_SPRINGWARE_E1 name is %s\n", aso_name);
}
```

### ■ See Also

- [gc\\_AlarmSourceObjectNameToID\(\)](#)



## gc\_AlarmSourceObjectName( )

**Name:** int gc\_AlarmSourceObjectName(metaeventp, aso\_name)

**Inputs:** METAEVENT \*metaeventp • pointer to metaevent  
char \*\*aso\_name • pointer to address of alarm source object name

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** GCAMS

**Mode:** synchronous

**Platform and** Springware: T-1/E-1, ISDN

**Technology:** DM3: T-1/E-1, ISDN  
IP (host-based stack)

### ■ Description

The **gc\_AlarmSourceObjectName( )** function returns the alarm source object (ASO) name of the current alarm event. The function is used after a GCEV\_ALARM event is received to retrieve the alarm source object name. The information retrieved by the function can be used in reports or for screen display.

Parameter	Description
<b>metaeventp</b>	points to metaevent
<b>aso_name</b>	points to the location where the pointer to the alarm source object name is to be stored. The <b>aso_name</b> cannot be NULL.

### ■ Termination Events

None

### ■ Cautions

- The **gc\_AlarmSourceObjectName( )** function can only be called for GCEV\_ALARM events.
- The data that is to be retrieved by the **gc\_AlarmSourceObjectName( )** function is valid only until the next event is requested.
- Do not overwrite the value that is returned in **aso\_name** as this points to private, internal GCAMS space.

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*.

All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

### ■ Example

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

    METAEVENT    metaevent;
    char          *aso_name;
    int           rc;

/*
-- This code assumes that the current event is GCEV_ALARM
-- event and that gc_GetMetaEvent() has already been called
-- to place the current event information into metaevent
*/

    rc = gc_AlarmSourceObjectName(&metaevent, &aso_name);
    if (rc < 0)
    {
        /* get and process the error */
    }
    else
    {
        printf("Alarm source object name is %s\n", aso_name);
    }
```

### ■ See Also

None

## `gc_AlarmSourceObjectNameToID( )`

**Name:** `int gc_AlarmSourceObjectNameToID(aso_name, aso_id)`

**Inputs:** `char *aso_name` • pointer to alarm source object name  
`unsigned long *aso_id` • pointer to destination for alarm source object ID

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** GCAMS

**Mode:** synchronous

**Platform and** Springware: T-1/E-1, ISDN

**Technology:** DM3: T-1/E-1, ISDN  
IP (host-based stack)

### ■ Description

The `gc_AlarmSourceObjectNameToID( )` function converts the alarm source object (ASO) name to the ASO ID. This function is used to obtain the ASO ID when the name of the alarm source object is known. The ASO ID can then be used in other functions that require the ASO ID as an input.

Parameter	Description
<code>aso_name</code>	points to the location of ASO name. The <code>aso_name</code> cannot be NULL.
<code>aso_id</code>	points to the destination where the alarm source object ID is returned. The <code>aso_id</code> cannot be NULL.

### ■ Termination Events

None

### ■ Cautions

None

### ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo( )` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

### ■ Example

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

unsigned long  aso_id;
int           rc;

rc = gc_AlarmSourceObjectNameToID("Springware E1", &aso_id);
if (rc < 0)
{
    /* get and process the error */
}
else
{
    printf("Alarm source object ID for Springware E1 is %d\n", aso_id);
}
```

### ■ See Also

- [gc\\_AlarmSourceObjectIDToName\(\)](#)

## `gc_AnswerCall()`

**Name:** `int gc_AnswerCall(crn, rings, mode)`

**Inputs:**

<code>CRN crn</code>	• call reference number
<code>int rings</code>	• number of rings before return
<code>unsigned long mode</code>	• async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** basic

**Mode:** asynchronous or synchronous

**Platform and** All†

**Technology:** †See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

The `gc_AnswerCall()` function indicates (to the originator) that the destination party is connected 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. Upon successful completion of the `gc_AnswerCall()` function, the call state changes from the Offered state or the Accepted state to the Connected state.

See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>crn</b>	call reference number
<b>rings</b>	<p>specifies the number of rings the protocol handler waits before notifying the calling entity. The valid values are as follows:</p> <ul style="list-style-type: none"> <li>• 0 to 14 – the call is answered after the specified number of rings</li> <li>• 15 or greater – the call will ring forever; this allows the application to accept calls without answering them</li> </ul> <p>For protocols not using rings, the <b>rings</b> parameter is ignored.</p>
<b>mode</b>	set to <code>EV_ASYNC</code> for asynchronous execution or to <code>EV_SYNC</code> for synchronous execution

### ■ Termination Events

`GCEV_ANSWERED`

indicates successful completion of the `gc_AnswerCall()` function, that is, the call was answered

**GCEV\_TASKFAIL**

indicates that the function failed. For more information, see the “Error Handling” section in the *Global Call API Programming Guide*.

**Note:** A GCEV\_DISCONNECTED event may be reported as an unsolicited event 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 the function fails.

■ **Errors**

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function for error information. If the GCEV\_TASKFAIL event is received, use the **gc\_ResultInfo()** function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

■ **Example**

```
#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)
 * 4. The event is determined to be a GCEV_OFFERED event
 */
int answer_call(void)
{
    CRN          crn;                /* call reference number */
    GC_INFO      gc_error_info;      /* GlobalCall error information data */
    /*
     * 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_ErrorInfo( &gc_error_info );
    printf ("Error: gc_AnswerCall() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
          metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
          gc_error_info.ccLibId, gc_error_info.ccLibName,
          gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}

/*
 * gc_AnswerCall() terminates with GCEV_ANSWERED event
 */
return (0);
}

```

#### ■ See Also

- [gc\\_AcceptCall\(\)](#)
- [gc\\_DropCall\(\)](#)
- [gc\\_WaitCall\(\)](#)

## **gc\_Attach()**

**Name:** int gc\_Attach(linedev, voiceh, mode)

**Inputs:** LINEDEV linedev                      • Global Call line device handle  
int voiceh                                      • voice device handle  
unsigned long mode                              • async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** media

**Mode:** asynchronous or synchronous

**Platform and** Springware: T-1/E-1

**Technology:** DM3: T-1/E-1, ISDN, Analog  
SS7

---

### ■ Description

**Note:** The **gc\_Attach()** function is deprecated in this software release. The **gc\_AttachResource()** function replaces the **gc\_Attach()** function and must be used instead.

The **gc\_Attach()** function attaches a voice resource to a line device.



## `gc_AttachResource( )`

**Name:** `int gc_AttachResource (linedev, resourcech, resourceattrp, retblkpp, resourcetype, mode)`

**Inputs:**

<code>LINEDEV linedev</code>	• Global Call line device handle
<code>int resourcech</code>	• voice or media device handle
<code>GC_PARM_BLK* resourceattrp</code>	• pointer to voice or media attribute information
<code>GC_PARM_BLK* retblkpp</code>	• pointer to address of <a href="#">GC_PARM_BLK</a> structure where voice or media attribute information is to be returned
<code>int resourcetype</code>	• type of resource
<code>unsigned long mode</code>	• async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** voice and media

**Mode:** asynchronous or synchronous

**Platform and Technology:** Springware: T-1/E-1

DM3: T-1/E-1, ISDN, Analog†

IP (host-based stack and embedded stack‡)

†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The `gc_AttachResource( )` function attaches a voice or media resource to a line device and provides optional capability exchange. By attaching the resource, an association is made between the line device and the voice or media resource channel. The resource channel specified by the device handle, **resourcech**, will be used to handle related Global Call functions requiring a voice or media resource for that line device.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>resourcech</b>	SRL device handle for the voice or media resource that is to be attached to the line device
<b>resourceattrp</b>	points to the <a href="#">GC_PARM_BLK</a> structure containing attributes of the voice or media resource being specified by <b>resourcech</b> . See the appropriate <i>Global Call Technology User's Guide</i> for information about how to specify resource attributes.
<b>retblkpp</b>	points to the address of <a href="#">GC_PARM_BLK</a> structure where voice or media attribute information is to be returned

Parameter	Description
<b>resourcetype</b>	type of resource. Possible values are: <ul style="list-style-type: none"> <li>• GC_VOICEDevice</li> <li>• GC_MEDIADevice</li> </ul>
<b>mode</b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

## ■ Termination Events

### GCEV\_ATTACH

indicates successful completion of the **gc\_AttachResource()** function.

### GCEV\_ATTACH\_FAIL

indicates that the **gc\_AttachResource()** function failed. For more information, see the appropriate *Global Call Technology User's Guide*.

## ■ Cautions

- DM3 voice resources may only be attached to DM3 network resources; that is, they cannot be attached to Springware devices. Similarly, Springware resources cannot be attached to DM3 devices.
- A single media resource, a single voice resource, or both a voice and media resource may be attached to a Global Call line device at any given time.
- The call control libraries process and consume voice or media events of interest to the call control library before they reach the application.
- The **gc\_AttachResource()** 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\_OpenEx()** function may be used to open, attach and route both the voice or media and the network resources.
- If this function is invoked for an unsupported technology, the function fails. The error value EGC\_UNSUPPORTED is returned when the **gc\_ErrorInfo()** function is used to retrieve the error code.
- If a protocol uses Global Tone Detection (GTD) tones for call analysis (assuming that GTD has not been disabled using the **gc\_SetParm()** function), all pre-existing tone definitions for that voice resource are deleted. If the application requires additional tones after the initial set of tones are loaded, they must be redefined after calling the **gc\_AttachResource()** function. The tone IDs cannot be in the range from 101-189.
- For protocols using GTD in any application that calls the **gc\_AttachResource()** function several times on the same device (for example, when using resource sharing), the overhead associated with redundant tone deletion and definition may be avoided by calling the **gc\_SetParm(ldev, GCPR\_LOADTONES, GCPV\_DISABLE)** function after the first call to the **gc\_AttachResource()** function. Afterward, the application must reverse the effect of the **gc\_SetParm()** function by issuing a **gc\_SetParm(ldev, GCPR\_LOADTONES, GCPV\_ENABLE)** function when the call is complete.
- When using call progress analysis, the **dx\_initcallp()** function must be called after calling the **gc\_AttachResource()** function. The **dx\_initcallp()** function initializes and activates PerfectCall call analysis on the channel and also adds all tones used in call analysis to the channel's GTD templates. Calling **dx\_initcallp()** before calling **gc\_AttachResource()** results

in the deletion of all tones and the possible disabling of PerfectCall call analysis by the latter function.

## ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function for error information. If the `GCEV_ATTACH_FAIL` event is received, use the `gc_ResultInfo()` function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxlib.h>
#include <gcclib.h>
#include <gcerr.h>

int attach(void)
{
    LINEDEV    ldev;                /* GlobalCall line device handle */
    int         voiceh;             /* Voice channel number */
    char        devname[50];
    GC_INFO     gc_error_info;      /* GlobalCall error information data */

    /*
     * Open line device for 1st network time slot on dtiB1
     */

    if (gc_OpenEx(&ldev, devname, EV_SYNC, &ldev) == GC_SUCCESS) {
        voiceh = dx_open("dxxB1C1", NULL);
        if (voiceh != -1) {
            if (gc_AttachResource(ldev, voiceh, NULL, NULL, GC_VOICEDevice,
                                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_AttachResource() error return as shown */
            gc_ErrorInfo(&gc_error_info);
            printf ("Error: gc_AttachResource() on device handle: 0x%lx, GC ErrorValue:
0x%hx - %s, CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                    ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                    gc_error_info.ccLibId, gc_error_info.ccLibName,
                    gc_error_info.ccValue, gc_error_info.ccMsg);
            return (gc_error_info.gcValue);
        }
    }
    else {
        /* Process dx_open() error */
    }
}
else {
    /* process error from gc_OpenEx() using gc_ErrorInfo() */
    gc_ErrorInfo(&gc_error_info);
    printf ("Error: gc_OpenEx() on devname: %s, GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s,
```

```
CC ErrorValue: 0x%lx - %s\n",
    devname, gc_error_info.gcValue, gc_error_info.gcMsg,
    gc_error_info.ccLibId, gc_error_info.ccLibName,
    gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}
return (0);
}
```

#### ■ See Also

- [gc\\_Close\(\)](#)
- [gc\\_Detach\(\)](#)
- [gc\\_GetNetworkH\(\)](#) (deprecated)
- [gc\\_GetResourceH\(\)](#)
- [gc\\_GetVoiceH\(\)](#) (deprecated)
- [gc\\_LoadDxParm\(\)](#)
- [gc\\_OpenEx\(\)](#)

## `gc_BlindTransfer()`

**Name:** `int gc_BlindTransfer(activecall, numberstr, makecallp, timeout, mode)`

**Inputs:**

<code>CRN activecall</code>	• call reference number for active call
<code>char *numberstr</code>	• consultation phone number
<code>GC_MAKECALL_BLK *makecallp</code>	• pointer to outbound (consultation) call information
<code>int timeout</code>	• time-out value
<code>unsigned long mode</code>	• async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** advanced call model

**Mode:** asynchronous or synchronous

**Platform and Technology:** Springware: T-1/E-1 (PDKRT only)†  
DM3: T-1/E-1†

†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The `gc_BlindTransfer()` function is used to initiate and complete a one-step transfer, that is, an unsupervised transfer.

For more information about unsupervised call transfers, see the *Global Call API Programming Guide*. See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>activecall</b>	call reference number for the original (active) call
<b>numberstr</b>	called party's (consultation call) telephone number. The number must be terminated with "\0". The maximum length is 32 digits.
<b>makecallp</b>	specifies the pointer to the <code>GC_MAKECALL_BLK</code> structure. The structure lists the parameters used to make the outbound call. See <a href="#">GC_MAKECALL_BLK</a> , on page 390 for more information.
<b>timeout</b>	time interval, in seconds, during which the call must be established or the 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; currently, <b>timeout</b> is supported only for the PDKRT library.
<b>mode</b>	set to <code>EV_ASYNC</code> for asynchronous execution or to <code>EV_SYNC</code> for synchronous execution

## ■ Termination Events

### GCEV\_BLINDTRANSFER

indicates that the call transfer was successfully initiated by the **gc\_BlindTransfer()** function.

### GCEV\_TASKFAIL

indicates that the function failed. For more information, see the “Error Handling” section in the *Global Call API Programming Guide*.

## ■ Cautions

- The **gc\_BlindTransfer()** function can be called only when the call to be transferred is in the Connected call state.
- After the successful return of the **gc\_BlindTransfer()** function, the application must call the **gc\_ReleaseCallEx()** function for the channel of the original active call to return the channel to the Null call state.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function for error information. If the GCEV\_TASKFAIL event is received, use the **gc\_ResultInfo()** function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <gcilib.h>
#include <gcerr.h>

#define MAX_CHAN 30          /* max. number of channels in system */
/*
 * Data structure which stores all information for each line
 */
static struct linebag {
    LINEDEV    ldev;          /* GlobalCall API line device handle */
    CRN        original_crn;   /* GlobalCall API call handle */
    CRN        consultation_crn; /* GlobalCall API call handle */
    int        blocked;        /* channel blocked/unblocked */
    int        networkh;       /* network handle */
} port[MAX_CHAN+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".
 * 3. A call has been established (original_crn) and is in connected state.
 */
int call_blindtransfer(int port_num)
{
    GC_INFO    gc_error_info;  /* GlobalCall error information data */
    /* Find info for this time slot, specified by 'port_num' */
    /* (assumes port_num is valid) */
    pline = port + port_num;
```

```

/* Transfer the original call to the consultation call at 993-3000 */
if (gc_BlindTransfer(pline->original_crn, "9933000", NULL, 0, EV_ASYNC) == -1)
{
    /* process error return as shown */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_BlindTransfer() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
           pline-> ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
           gc_error_info.ccLibId, gc_error_info.ccLibName,
           gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}

/* The gc_ReleaseCallEx() function must be called next on both the
/* consultation_crn and the original_crn to return the local
/* channels to the NULL state. */
return (0);
}

```

#### ■ See Also

- [gc\\_CompleteTransfer\(\)](#)
- [gc\\_SwapHold\(\)](#)
- [gc\\_SetUpTransfer\(\)](#)

## gc\_CallAck( )

**Name:** int gc\_CallAck(crn, callack\_blkp, mode)

**Inputs:**

CRN crn	• call reference number
GC_CALLACK_BLK *callack_blkp	• 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

**Category:** optional call handling

**Mode:** asynchronous or synchronous

**Platform and Technology:** Springware: T-1/E-1†, ISDN†

DM3: ISDN†

SS7†

IP (host-based stack)†

†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The **gc\_CallAck( )** function indicates (to the originator) call reception and optionally takes action or retrieves information from the network about the incoming call. This function is used when the call is in the Offered state (that is, 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, such as retrieving additional DDI (DNIS) digits, are available to all of the supported technologies. When the **gc\_CallAck( )** function is used to request additional DDI digits, use the **gc\_GetCallInfo( )** function to retrieve the DDI digits.

See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>crn</b>	call reference number
<b>callack_blkp</b>	points to the GC_CALLACK_BLK structure where the type field specifies the type of service requested by the <b>gc_CallAck( )</b> function. See <a href="#">GC_CALLACK_BLK</a> , on page 381 for further information.
<b>mode</b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution. When using ISDN protocols and the type field in the <a href="#">GC_CALLACK_BLK</a> data structure is set to GCAK_SERVICE_ISDN, this <b>mode</b> parameter must be set to EV_SYNC.

For example, to use the **gc\_CallAck( )** function to collect four DDI digits, set:



- `callack_blkp->type=GCACK_SERVICE_INFO`
- `callack_blkp->info.info_type=DESTINATION_ADDRESS`
- `callack_blkp->info.info_len=4`

Table 2 shows the values that are supported for the type field.

**Table 2. Possible Values for the type Field in GC\_CALLACK\_BLK**

type Value	Description
GCACK_SERVICE_INFO	Acknowledge the call and request more information. This type should be used instead of GCACK_SERVICE_DNIS (see below) to get more digits. <b>Note:</b> If this value is used for type, then the 'info' structure must be used in the service union. See Table 3 for information about the 'info' structure.
GCACK_SERVICE_PROC	Acknowledge that all the information has been received and the call is proceeding. This type should be used for acknowledging the call, for example, CALL_PROCEEDING for ISDN protocols, instead of using the GCACK_SERVICE_ISDN (see below) parameter.
GCACK_SERVICE_DNIS	Request retrieval of additional DNIS digits. <b>Note:</b> This type is not supported for all call libraries. It is recommended that GCACK_SERVICE_PROC be used instead.
GCACK_SERVICE_ISDN	Send the first response to an incoming call (ISDN only).

Table 3 describes the fields in the 'info' structure used when GCACK\_SERVICE\_INFO is specified.

**Table 3. Fields in the 'info' Structure for GCACK\_SERVICE\_INFO**

Field	Description
info_type	Type of information requested. The possible values are: <ul style="list-style-type: none"> <li>• DESTINATION_ADDRESS – request more DNIS</li> <li>• ORIGINATION_ADDRESS – request more ANI</li> </ul>
info_len	Length of the info requested (typically number of digits)

## ■ Termination Events

### GCEV\_ACKCALL

indicates that the `gc_CallAck()` function was successful, that is, the requested call information was retrieved. See the appropriate *Global Call Technology User's Guide* to determine if this event is supported.

### GCEV\_MOREINFO

this event is received if the GCACK\_SERVICE\_INFO type is used. This event indicates the status of the information (typically digits) that was requested. The `extevtdatp` field of the METAEVENT contains the status of the information. Use the `gc_ResultValue()` function to get the result value that indicates if the requested number of digits is available, more digits are available, or no digits are available. Table 11, "Result Values for GCEV\_MOREINFO", on page 233 indicates the valid values.

#### **GCEV\_CALLPROC**

this event is received if the `GCACK_SERVICE_PROC` type is used. This event indicates that the all the necessary information for the call has been received and the remote side has been notified that the call is now proceeding.

#### **GCEV\_TASKFAIL**

indicates that the function failed. For more information, see the “Error Handling” section in the *Global Call API Programming Guide*.

**Note:** When using the ISDN call control library, 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`.

### ■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value `EGC_UNSUPPORTED` will be the Global Call value returned when the [gc\\_ErrorInfo\(\)](#) function is used to retrieve the error code.

### ■ Errors

If this function returns `<0` to indicate failure, use the ***gc\_ErrorInfo()*** function for error information. If the `GCEV_TASKFAIL` event is received, use the ***gc\_ResultInfo()*** function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

### ■ Example

```
#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 */
    GC_INFO gc_error_info;    /* GlobalCall error information data */

    /*
     * Do the following:
     * 1. Get called party number using gc_GetCallInfo() 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_INFO;
callack.service.info.info_type = DESTINATION_ADDRESS;
callack.service.info.info_len = GCDG_NDIGIT; /* Set to 3 */
if (gc_CallAck(crn, &callack, EV_ASYNC) != GC_SUCCESS) {
    /* process error return as shown */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_CallAck() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
        metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
        gc_error_info.ccLibId, gc_error_info.ccLibName,
        gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}
/*
 * Now collect the remaining digits.
 */
if (gc_GetCallInfo(crn, DESTINATION_ADDRESS, dnis_buf) != GC_SUCCESS) {
    /* process error return as shown */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_GetCallInfo() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
        metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
        gc_error_info.ccLibId, gc_error_info.ccLibName,
        gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}

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

```

#### ■ See Also

- [gc\\_AcceptCall\(\)](#)
- [gc\\_AnswerCall\(\)](#)
- [gc\\_GetCallInfo\(\)](#)
- [gc\\_GetDNIS\(\)](#) (deprecated)
- [gc\\_WaitCall\(\)](#)

## gc\_CallProgress()

**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

**Category:** interface specific

**Mode:** synchronous

**Platform and** Springware: ISDN†

**Technology:** †See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

The **gc\_CallProgress()** function notifies the network that provides information about the progress of a call. 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 in-band 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 in-band Special Information Tone (SIT) or call progress tone is sent in the network direction.

See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>crn</b>	call reference number
<b>indicator</b>	call progress indicator: <ul style="list-style-type: none"><li>• 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.</li><li>• 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.</li></ul>

### ■ Termination Events

None

## ■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value `EGC_UNSUPPORTED` will be the Global Call value returned when the `gc_ErrorInfo()` function is used to retrieve the error code.

## ■ Errors

If this function returns `<0` to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ 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) has been
 *    called to convert the event into metaevent.
 * 4. a GCEV_OFFERED event has been detected.
 */

#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   ldev;           /* Line device */
    GC_INFO    gc_error_info; /* GlobalCall error information data */

    if(gc_CRN2LineDev(crn, &ldev) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_CRN2LineDev() on crn: 0x%lx, GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                crn, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    if(gc_CallProgress(crn, indicator) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_CallProgress() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
```

```
        gc_error_info.ccValue, gc_error_info.ccMsg);  
    return (gc_error_info.gcValue);  
}  
return(0);  
}
```

■ **See Also**

- [gc\\_DropCall\(\)](#)
- [gc\\_WaitCall\(\)](#)

## `gc_CCLibIDToName()`

**Name:** `int gc_CCLibIDToName(cclibid, lib_name)`

**Inputs:** `int cclibid`                      • ID code of library  
`char **lib_name`                      • pointer to address of library name

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gcLib.h`  
`gcerr.h`

**Category:** library information

**Mode:** synchronous

**Platform and Technology:** All

---

### ■ Description

The `gc_CCLibIDToName()` function converts call control library ID to name of the 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
<code>cclibid</code>	identification number of call control library. If a library name is not associated with this parameter, then the library name will be NULL.
<code>lib_name</code>	points to the location where the name of the call control library associated with the <code>cclibid</code> parameter is stored. See <code>gc_CCLibNameToID()</code> for valid call control library names.

### ■ Termination Events

None

### ■ Cautions

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

### ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

■ **Example**

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

int cclibid_to_name(int cclibid, char **lib_name)
{
    GC_INFO      gc_error_info;    /* GlobalCall error information data */

    if (gc_CCLibIDToName(cclibid, lib_name) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_CCLibIDToName(), cclibid: 0x%lx, GC ErrorValue: 0x%hx - %s, CCLibID:
%i - %s, CC ErrorValue: 0x%lx - %s\n",
                cclibid, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    return(0);
}
```

■ **See Also**

- [\*\*\*gc\\_CCLibNameToID\(\)\*\*\*](#)



## `gc_CCLibNameToID()`

**Name:** `int gc_CCLibNameToID(lib_name, cclibidp)`

**Inputs:** `char *lib_name` • pointer to 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

**Platform and Technology:** All

### ■ 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
<b>lib_name</b>	name of the call control library whose library ID is to be retrieved. Valid call control library names are: <ul style="list-style-type: none"> <li>• “GC_ALL_LIB” – All call control libraries</li> <li>• “GC_ANAPI_LIB” – ANAPI call control library</li> <li>• “GC_CUSTOM1_LIB” – Custom call control library 1</li> <li>• “GC_CUSTOM2_LIB” – Custom call control library 2</li> <li>• “GC_DM3CC_LIB” – DM3CC call control library</li> <li>• “GC_ICAPI_LIB” – ICAPI call control library</li> <li>• “GC_IPM_LIB” – IP_Media call control library</li> <li>• “GC_ISDN_LIB” – ISDN call control library</li> <li>• “GC_PDKRT_LIB” – PDKRT call control library</li> <li>• “GC_SS7_LIB” – SS7 call control library</li> </ul>
<b>cclibidp</b>	points to identification code of call control library

### ■ Termination Events

None

### ■ Cautions

If an invalid **lib\_name** is passed to the `gc_CCLibNameToID()` function, the function will **not** return <0 to indicate failure; the information stored in **cclibidp** will remain unchanged (for example, 0).

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

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

int cclibName_to_id(char *lib_name, int *cclibidp)
{
    GC_INFO          gc_error_info;    /* GlobalCall error information data */

    if (gc_CCLibNameToID(lib_name, cclibidp) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_CCLibNameToID(), libname: %s, GC ErrorValue: 0x%x - %s, CCLibID: %i\n",
            lib_name, gc_error_info.gcValue, gc_error_info.gcMsg,
            gc_error_info.ccLibId, gc_error_info.ccLibName,
            gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    return(0);
}
```

## ■ See Also

- [gc\\_CCLibIDToName\(\)](#)

## `gc_CCLibStatus()`

**Name:** `int gc_CCLibStatus(cclib_name, cclib_infop)`

**Inputs:** `char *cclib_name`      • name of call control library  
           `int *cclib_infop`        • status of call control library

**Returns:** 0 if successful  
           <0 if failure

**Includes:** `gcclib.h`  
               `gcerr.h`

**Category:** library information

**Mode:** synchronous

**Platform and** All

**Technology:**

### ■ Description

**Note:** The `gc_CCLibStatus()` function is deprecated in this release. The suggested equivalent is `gc_CCLibStatusEx()`.

The `gc_CCLibStatus()` function retrieves the status of a single call control library specified by the **cclib\_name** parameter. The status of a library can be available, configured, or failed. This status information is stored in **\*cclib\_infop**.

Parameter	Description
<b>cclib_name</b>	name of the call control library. See <code>gc_CCLibNameToID()</code> for valid call control library names. The string must be set to one of these names and terminated by a NULL.
<b>cclib_infop</b>	points to location of status information. The status information is a bitmask with an available or configured mask set (these masks are mutually exclusive) and/or a failed mask: <ul style="list-style-type: none"> <li>• <code>GC_CCLIB_AVAILABLE</code> – available library (started successfully)</li> <li>• <code>GC_CCLIB_CONFIGURED</code> – configured library</li> <li>• <code>GC_CCLIB_FAILED</code> – library failed to start</li> </ul>

### ■ Termination Events

None

### ■ Cautions

If an invalid **cclib\_name** is passed to the `gc_CCLibStatus()` function, the function will **not** return <0 to indicate failure; the information stored in **cclib\_infop** will remain unchanged (for example, 0).

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
#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 */
    GC_INFO      gc_error_info;      /* GlobalCall error information data */

    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_AVAILABLE) ? "yes" : "no");
        printf("    failed:     %s\n",
            (lib_status & GC_CCLIB_FAILED) ? "yes" : "no");
    } else {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_CCLibStatus(), libname: %s, GC ErrorValue: 0x%x - %s, CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
            lib_name, gc_error_info.gcValue, gc_error_info.gcMsg,
            gc_error_info.ccLibId, gc_error_info.ccLibName,
            gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    return(0);
}
```

## ■ See Also

- [gc\\_CCLibStatusEx\(\)](#)
- [gc\\_Start\(\)](#)

## `gc_CCLibStatusAll()`

**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:** `gcclib.h`  
`gcerr.h`

**Category:** library information

**Mode:** synchronous

**Platform and Technology:** All

### ■ Description

**Note:** The `gc_CCLibStatusAll()` function is deprecated in this software release. The suggested equivalent is `gc_CCLibStatusEx()`.

The `gc_CCLibStatusAll()` function retrieves the status of all call control libraries. Information returned includes the number and names of the available, configured, and failed call control libraries. See `gc_CCLibNameToID()` for valid call control library names. The Global Call library is not a call control library and is therefore not counted.

Parameter	Description
<code>cclib_status</code>	points to the <code>GC_CCLIB_STATUS</code> structure, which contains the library status information. See <code>GC_CCLIB_STATUS</code> , on page 384 for more information.

### ■ Termination Events

None

### ■ Cautions

- If any of the `num_*` fields is 0, then the corresponding `*libraries` field is NULL; for example, 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 Global Call data space.

### ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology

specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

### ■ Example

```
#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 */
    GC_INFO      gc_error_info; /* GlobalCall error information data */

    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_ErrorInfo( &gc_error_info );
        printf ("Error: gc_CCLibStatusAll(), GC ErrorValue: 0x%x - %s, CCLibID: %i - %s, CC
ErrorValue: 0x%x - %s\n",
               gc_error_info.gcValue, gc_error_info.gcMsg,
               gc_error_info.ccLibId, gc_error_info.ccLibName,
               gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    return(0);
}
```

### ■ See Also

- [gc\\_CCLibStatusEx\(\)](#)
- [gc\\_Start\(\)](#)

## gc\_CCLibStatusEx( )

**Name:** int gc\_CCLibStatusEx(cclib\_name, cclib\_infop)

**Inputs:** char \*cclib\_name                      • name of call control library  
void \*cclib\_infop                      • status of call control library

**Returns:** 0 if successful  
<0 if failure

**Includes:** gcclib.h  
gcerr.h

**Category:** library information

**Mode:** synchronous

**Platform and** All

**Technology:**

### ■ Description

The **gc\_CCLibStatusEx( )** function retrieves call control library status. The status retrieved can be for a single call control library, specified by the **cclib\_name** parameter, or the status of all libraries. The status of a library can be available, configured, or failed. This status information is stored in a location pointed to by **cclib\_infop**. The **cclib\_infop** parameter points to an integer location where the status information is stored for the single library specified. If the status for all the libraries is specified, then the **cclib\_infop** points to a [GC\\_CCLIB\\_STATUSALL](#) structure. The current functionality retrieves the status of one library at a time. This functionality will be expanded to provide the status of all the supported libraries if requested. If the status of all the libraries is to be retrieved, the GC\_ALL\_LIB name must be specified and a pointer to a GC\_CCLIB\_STATUSALL structure must be passed by the **cclib\_infop** parameter. The status will be returned in this structure.

Parameter	Description
<b>cclib_name</b>	call control library name. See <a href="#">gc_CCLibNameToID( )</a> for valid call control library names. The string must be set to one of these names in uppercase and terminated by a NULL.
<b>cclib_infop</b>	points to an integer location for status information. The possible states are: <ul style="list-style-type: none"> <li>• GC_CCLIB_AVAILABLE – available library (started successfully)</li> <li>• GC_CCLIB_CONFIGURED – configured library</li> <li>• GC_CCLIB_FAILED – library failed to start</li> </ul> <p>If the status of all the libraries is to be retrieved, that is, if GC_ALL_LIB is specified by the <b>cclib_name</b> parameter, the application must pass the pointer to a <a href="#">GC_CCLIB_STATUSALL</a> structure. The status of all the libraries will be returned in this structure.</p>

## ■ Termination Events

None

## ■ Cautions

If an invalid **cclib\_name** is passed to the **gc\_CCLibStatusEx( )** function, the function will **not** return <0 to indicate failure; the information stored in **cclib\_infolp** will remain unchanged (for example, 0).

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <gcclib.h>
#include <gcerr.h>

#define MAX_STRING_SIZE 100

int print_all_cclibs_status(void)
{
    int i;
    char str[MAX_STRING_SIZE], str1[MAX_STRING_SIZE];
    GC_CCLIB_STATUSALL cclib_status_all;
    GC_INFO gc_error_info; /* GlobalCall error information data */

    if (gc_CCLibStatusEx("GC_ALL_LIB", &cclib_status_all) != GC_SUCCESS) {
        /* error handling */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_CCLibStatusEx(), lib_name: %s, GC ErrorValue: 0x%hx - %s, CCLibID: %i\n",
            cclib_status_all.lib_name, gc_error_info.gcValue, gc_error_info.gcMsg,
            gc_error_info.ccLibId, gc_error_info.ccLibName);
        return (gc_error_info.gcValue);
    }

    strcpy(str, " Call Control Library Status:\n");

    for (i = 0; i < GC_TOTAL_CCLIBS; i++) {
        switch (cclib_status_all.cclib_state[i].state) {
            case GC_CCLIB_CONFIGURED:
                sprintf(str1, "%s - configured\n",
                    cclib_status_all.cclib_state[i].name);
                break;

            case GC_CCLIB_AVAILABLE:
                sprintf(str1, "%s - available\n",
                    cclib_status_all.cclib_state[i].name);
                break;
        }
    }
}
```



```

        case GC_CCLIB_FAILED:
            sprintf(str1, "%s - is not available for use\n",
                    cclib_status_all.cclib_state[i].name);
            break;

        default:
            sprintf(str1, "%s - unknown CCLIB status\n",
                    cclib_status_all.cclib_state[i].name);
            break;
    }
    strcat(str, str1);
}
printf(str);
return (0);
}

```

#### ■ See Also

- [gc\\_Start\(\)](#)

## gc\_Close()

**Name:** int gc\_Close(linedev)

**Inputs:** LINEDEV linedev

• Global Call line device handle

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** system controls and tools

**Mode:** synchronous

**Platform and** All†

**Technology:** †See the *Global Call Technology User's Guides* for additional information.

---

### ■ 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 are not affected. The **gc\_Close()** function should be issued while the line device is in the Null call state. The Null call state occurs immediately after a call to **gc\_OpenEx()**, **gc\_ResetLineDev()**, or **gc\_ReleaseCallEx()**.

See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>linedev</b>	Global Call line device to close

### ■ Termination Events

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 media resource is attached to the **linedev** device, the media resource will be closed by the Global Call API. To keep the media resource open for other operations, use the **gc\_Detach()** function to detach the media resource from the Global Call device before issuing the **gc\_Close()** function.
- The **gc\_Close()** function must not be called from a Standard Runtime Library (SRL) handler. If the **gc\_Close()** function is called from an SRL handler, the application may hang or SRL functionality may fail.

## ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

```
#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 */
    GC_INFO gc_error_info; /* GlobalCall error information data */

    /* 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_ErrorInfo(&gc_error_info);
        printf("Error: gc_Close() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
            ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
            gc_error_info.ccLibId, gc_error_info.ccLibName,
            gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    return(0);
}
```

## ■ See Also

- [gc\\_AttachResource\(\)](#)
- [gc\\_Attach\(\)](#) (deprecated)
- [gc\\_Detach\(\)](#)
- [gc\\_OpenEx\(\)](#)

## gc\_CompleteTransfer( )

**Name:** int gc\_CompleteTransfer(callonhold, consultationcall, mode)

**Inputs:**

CRN callonhold	• call reference number for call on hold
CRN *consultationcall	• call reference number for consultation call
unsigned long mode	• async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** advanced call model

**Mode:** asynchronous or synchronous

**Platform and** Springware: T-1/E-1 (PDKRT only)†

**Technology:** †See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The **gc\_CompleteTransfer( )** function completes the transfer of a call in a supervised transfer. A supervised transfer means that the party receiving the transferred call is consulted by the party transferring the call, before the transfer is completed. This function must be called only after the transferred call has been delivered and the channel is in the Dialtone state or the OnHoldPendingTransfer state.

This function is supported only under the PDKRT call control library. See the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>callonhold</b>	call reference number for the call on hold, that is, the call pending transfer
<b>consultationcall</b>	call reference number for the consultation call
<b>mode</b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

### ■ Termination Events

GCEV\_COMPLETETRANSFER

indicates that the **gc\_CompleteTransfer( )** function was successful, that is, the transfer was completed

GCEV\_TASKFAIL

indicates that the function failed. For more information, see the “Error Handling” section in the *Global Call API Programming Guide*.

## ■ Cautions

- In synchronous mode, after the successful return of the `gc_CompleteTransfer()` function, the application must call the `gc_ReleaseCallEx()` function for both channels (the consultation call and the call that was transferred) to return the channels to the Null state.
- In asynchronous mode, after receiving the `GCEV_COMPLETETRANSFER` event, the application must call the `gc_ReleaseCallEx()` function for both channels (the consultation call and the call that was transferred) to return the channels to the Null state.

## ■ Errors

If this function returns `<0` to indicate failure, use the `gc_ErrorInfo()` function for error information. If the `GCEV_TASKFAIL` event is received, use the `gc_ResultInfo()` function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

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

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

/*
 * Data structure which stores all information for each line
 */
static struct linebag {
    LINEDEV    ldev;          /* GlobalCall API line device handle */
    CRN        original_crn;   /* GlobalCall API call handle */
    CRN        consultation_crn; /* GlobalCall API call handle */
    int        blocked;        /* channel blocked/unblocked */
    int        networkh;       /* network handle */
} port[MAX_CHAN+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".
 * 3. A call has been established (original_crn) and is in connected state.
 * 4. The gc_SetupTransfer() has been called successfully
 *    to initiate the transfer.
 */
int call_completetransfer(int port_num)
{
    GC_INFO    gc_error_info; /* GlobalCall error information data */

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

    /* Complete the transfer of the consultation call
     * to the original call */
    if (gc_CompleteTransfer(pline->original_crn, pline->consultation_crn,
        EV_ASYNC) == -1)
    {
        /* process error return as shown */
    }
}
```

```
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_CompleteTransfer() on device handle: 0x%lx, GC ErrorValue: 0x%hx -
%s, CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                pline->ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    /* The gc_ReleaseCallEx() function must be called next on both
    /* the consultation_crn and the original_crn to return
    /* the local channels to the NULL state. */

    return (0);
}
```

■ **See Also**

- [gc\\_SetUpTransfer\(\)](#)
- [gc\\_BlindTransfer\(\)](#)
- [gc\\_SwapHold\(\)](#)

## `gc_CRN2LineDev()`

**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 controls and tools

**Mode:** synchronous

**Platform and Technology:** All

---

### ■ Description

The `gc_CRN2LineDev()` function is a utility function that maps a call reference number (CRN) to its line device ID. This function returns the line device identification associated with the specified CRN.

Parameter	Description
<code>crn</code>	call reference number
<code>linedevp</code>	points to the location where the output LINEDEV identification code will be stored. The line device is created when the function <code>gc_OpenEx()</code> is called.

### ■ Termination Events

None

### ■ Cautions

A CRN is no longer valid once a `gc_ReleaseCallEx()` or `gc_ResetLineDev()` function has been issued.

### ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcErr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccErr.h` or `isdnErr.h` file for the ISDN call control library).

## ■ Example

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

int crn_to_linedev(CRN crn, LINEDEV *ldevp)
{
    GC_INFO    gc_error_info;    /* GlobalCall error information data */

    if (gc_CRN2LineDev(crn, ldevp) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_CRN2LineDev() on crn: 0x%x, GC ErrorValue: 0x%x - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
                crn, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    return(0);
}
```

## ■ See Also

None



**gc\_Detach()****Name:** int gc\_Detach(linedev, voiceh, mode)

**Inputs:** LINEDEV linedev      • Global Call line device handle  
 int voiceh      • media device handle  
 unsigned long mode      • async or sync

**Returns:** 0 if successful  
<0 if failure**Includes:** gclib.h  
gcerr.h**Category:** voice and media**Mode:** asynchronous or synchronous**Platform and** Springware: T-1/E-1†**Technology:** DM3: T-1/E-1†, ISDN, Analog  
SS7

IP (host-based stack and embedded stack†)

†See the *Global Call Technology User's Guides* for additional information.■ **Description**

The **gc\_Detach()** function is used to detach a voice or media 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\_AttachResource()** function.

When a **gc\_Close()** function closes a line device, any attached voice or media resource is closed automatically. To keep the voice or media device open, first issue a **gc\_Detach()** function and then issue the **gc\_Close()** function. This will disassociate the voice or media device from the line device.

See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>voiceh</b>	SRL device handle of the voice or media resource to be detached from the call control line device
<b>mode</b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

■ **Termination Events**

GCEV\_DETACH

indicates successful completion of the **gc\_Detach()** function.

**GCEV\_DETACH\_FAIL**

indicates that the **gc\_Detach()** function failed. For more information, see the appropriate *Global Call Technology User's Guide*.

■ **Cautions**

- The **gc\_Detach()** function does **not** perform any routing or unrouting functions. For technologies where time slot routing is applicable, 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 returned when the **gc\_ErrorInfo()** function is used to retrieve the error code.
- If the **gc\_Detach()** function is called during call setup, the application may receive an unexpected event. In addition, an unexpected event may be received if a call to the **gc\_Detach()** function is followed by a call to **gc\_Close()**. These unexpected events can be ignored.
- In applicable technologies, **gc\_Detach()** does not unload tones.
- Global Call functionality requiring a voice resource, such as **gc\_MakeCall()** or **gc\_WaitCall()** on R2 technologies, will fail if there are no voice resources attached.

■ **Errors**

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function for error information. If the **GCEV\_DETACH\_FAIL** event is received, use the **gc\_ResultInfo()** function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

■ **Example**

```
#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)
{
    GC_INFO          gc_error_info;    /* GlobalCall error information data */
    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_ErrorInfo( &gc_error_info );
            printf ("Error: gc_Detach() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s",

```

```

        CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
        ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
        gc_error_info.ccLibId, gc_error_info.ccLibName,
        gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}

/*
 * 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 */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_GetVoiceH() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
        CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
        ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
        gc_error_info.ccLibId, gc_error_info.ccLibName,
        gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}
return (0);
}

```

#### ■ See Also

- [gc\\_AttachResource\(\)](#)
- [gc\\_Close\(\)](#)
- [gc\\_OpenEx\(\)](#)

## gc\_DropCall( )

**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

**Mode:** asynchronous or synchronous

**Platform and** All†

**Technology:** †See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The **gc\_DropCall( )** function disconnects a call specified by the call reference number (CRN) and enables inbound calls to be detected internally to Global Call on the line device.

The application will not be notified of any pending call until after the **gc\_ReleaseCallEx( )** function is issued. However, for some technologies, a GCEV\_OFFERED event may be generated after **gc\_DropCall( )** is issued and before the call is released. The application must allow for this possibility and be able to handle the event.

The **gc\_DropCall( )** function changes the current active call state (Accepted, Answered, Alerting, Connected, Detected, or Offered) to the Idle state.

See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>crn</b>	call reference number
<b>cause</b>	indicates reason for disconnecting or rejecting a call. See Table 4 for a list of possible causes, and see the appropriate <i>Global Call Technology User's Guide</i> for valid and/or additional causes for your specific technology.
<b>mode</b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

**Table 4. `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
† See the appropriate <i>Global Call Technology User's Guide</i> for valid and/or additional causes for your specific technology.	

### ■ Termination Events

#### GCEV\_DROPCALL

indicates that the **`gc_DropCall()`** function was successful, that is, the call was dropped.

#### GCEV\_TASKFAIL

indicates that the function failed. For more information, see the “Error Handling” section in the *Global Call API Programming Guide*.

**Note:** A GCEV\_DISCONNECTED event may be reported to the application as an unsolicited event after the **`gc_DropCall()`** function is issued.

### ■ Cautions

- The **`gc_DropCall()`** function does not release a CRN. Therefore, the **`gc_ReleaseCallEx()`** 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.
- You must use the **`dx_stopch()`** function to terminate any application-initiated voice functions, such as **`dx_play()`** or **`dx_record()`**, before calling **`gc_DropCall()`**.
- 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, from the Accepted state. If a forced release is attempted, the function will fail and an error will be returned. To recover, the application should issue a **`gc_AnswerCall()`** function followed by **`gc_DropCall()`** and **`gc_ReleaseCallEx()`** functions. When using the ICAPI call control library, see the *Global Call Country Dependent Parameters (CDP) Reference* for protocol specific limitations. However, any time 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; see the appropriate *Global Call Technology User's Guide* for valid causes for your specific technology.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function for error information. If the GCEV\_TASKFAIL event is received, use the **gc\_ResultInfo()** function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
#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)
{
    GC_INFO      gc_error_info;    /* GlobalCall error information data */

    if (gc_DropCall(crn, GC_NORMAL_CLEARING, EV_ASYNC) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_DropCall() device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    /*
     * gc_DropCall() is terminated by the GCEV_DROP_CALL event.
     * Application must then release the call using gc_ReleaseCallEx().
     */
    return (0);
}
```

## ■ See Also

- [gc\\_MakeCall\(\)](#)
- [gc\\_ReleaseCallEx\(\)](#)
- [gc\\_WaitCall\(\)](#)

## `gc_ErrorInfo()`

**Name:** `int gc_ErrorInfo(*a_Infop)`

**Inputs:** `GC_INFO *a_Infop` • pointer to the `GC_INFO` data structure

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

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** system controls and tools

**Mode:** synchronous

**Platform and** All

**Technology:**

---

### ■ Description

The `gc_ErrorInfo()` function provides error information about a failed function. Calling the `gc_ErrorInfo()` function is equivalent to calling the `gc_ErrorValue()`, `gc_ResultMsg()`, and `gc_CCLibIDToName()` functions. To retrieve the information, the `gc_ErrorInfo()` function must be called immediately after the Global Call function failed. The `gc_ErrorInfo()` function returns an error value associated directly with the Global Call and call control library.

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 (for example, ISDN time-out errors differ from those in an R2 MF protocol). Each of these call control library time-out errors are mapped to the time-out error, `EGC_TIMEOUT`. However, the specific time-out error detected by the call control library is available through the `ccValue` field in the `GC_INFO` data structure. To aid in debugging, both the `gcMsg` and the `ccMsg` value fields in the `GC_INFO` data structure should be retrieved.

Parameter	Description
<code>a_Infop</code>	points to the <code>GC_INFO</code> structure where information about the error is contained.

### ■ Termination Events

None

### ■ Cautions

- The `gc_ErrorInfo()` function can only be called in the **same** thread in which the routine that had the error was called. The `gc_ErrorInfo()` function cannot be called to retrieve error information for a function that returned error information in another thread.

- Do not overwrite the message space pointed to by any of the char \* in the [GC\\_INFO](#) data structure as these point to private internal space.
- The lifetime of the strings pointed to by the [GC\\_INFO](#) data structure is from the time the **gc\_ErrorInfo()** function returns to the time the next Global Call function is called.

## ■ Errors

The **gc\_ErrorInfo()** function should not be called recursively if it returns <0 to indicate failure. A failure return generally indicates that **a\_Infop** is NULL.

## ■ Example

```
/*
-- This function can be called anytime an error occurs
-- Not shown is a GlobalCall function which fails and calls this function
-- This procedure prints error information to the console with no other side effects
*/

void PrintGC_INFO(GC_INFO *a_Infop);

void PrintErrorInfo(void)
{
    int             retCode;
    GC_INFO         t_Infop;
    retCode = gc_ErrorInfo(&t_Infop);
    if (retCode == GC_SUCCESS) {
        printf("gc_ErrorInfo() successfully called\n");
        PrintGC_INFO(&t_Infop);
    } else {
        printf("gc_ErrorInfo() call failed\n");
    }
}

/*
-- This function is called to print GC_INFO to the system console
-- Typically it would be called after a call to gc_ErrorInfo()
-- or gc_ResultInfo() to print the resulting GC_INFO data structure
*/

void PrintGC_INFO(GC_INFO *a_Infop)
{
    printf("a_Infop->gcValue = 0x%x\n", a_Infop->gcValue);
    printf("a_Infop->gcMsg = %s\n", a_Infop->gcMsg);
    printf("a_Infop->ccLibId = %d\n", a_Infop->ccLibId);
    printf("a_Infop->ccLibName = %s\n", a_Infop->ccLibName);
    printf("a_Infop->ccValue = 0x%x\n", a_Infop->ccValue);
    printf("a_Infop->ccMsg = %s\n", a_Infop->ccMsg);
    printf("a_Infop->additionalInfo = %s\n", a_Infop->additionalInfo);
    printf("Enter <CR> to continue: ");
    getchar();
}
```

## ■ See Also

- [gc\\_ErrorValue\(\)](#) (deprecated)
- [gc\\_ResultInfo\(\)](#)
- [gc\\_ResultMsg\(\)](#) (deprecated)
- [gc\\_CCLibIDToName\(\)](#)



## `gc_ErrorValue()`

**Name:** `int gc_ErrorValue(gc_errorp, cclibidp, cclib_errorp)`

**Inputs:**

- `int *gc_errorp` • location to store Global Call 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:** `gcilib.h`  
`gcerr.h`

**Category:** system controls and tools

**Mode:** synchronous

**Platform and** All†

**Technology:** †See the *Global Call Technology User's Guides* for additional information.

### ■ Description

**Note:** The `gc_ErrorValue()` function is deprecated in this software release. The suggested equivalent is `gc_ErrorInfo()`.

The `gc_ErrorValue()` function gets an error value/failure reason code associated with the last Global Call function call. To retrieve an error, this function must be called immediately after a Global Call function failed. This function returns the Global Call error code, `*gc_errorp`, as well as the lower level error code associated directly with the call control library, `*cclib_errorp`. The Global Call 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
<code>gc_errorp</code>	points to the location where the Global Call error code will be stored
<code>cclibidp</code>	points to the location to store the identification number of the call control library where the error occurred
<code>cclib_errorp</code>	points to the location to store the call control library error description that is uniquely associated to its own library

## ■ Termination Events

None

## ■ Cautions

- To aid in debugging, both the **gc\_errorp** and the **cclib\_errorp** values should be retrieved.
- The **gc\_ErrorValue()** function can be called only in the **same** thread in which the routine in error was called. The function is “thread-safe” and it gets the value from thread-safe memory. The **gc\_ErrorValue()** function cannot be called to retrieve the error value for a function that returned an error in another thread.

## ■ Errors

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

## ■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <gcilib.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);
        /* GlobalCall errors will return an integer for the print error value. */
        printf("GlobalCall error 0x%lx - %s\n", gc_error, msg);
        gc_ResultMsg( cclibid, cc_error, &msg);
        gc_CCLibIDToName(cclibid, &lib_name);
        printf("%s library had error 0x%lx - %s\n", lib_name, cc_error, msg);
    } else {
        printf("Could not get error value\n");
    }
}
```

## ■ See Also

- [\*\*gc\\_ErrorInfo\(\)\*\*](#)
- [\*\*gc\\_ResultInfo\(\)\*\*](#)
- [\*\*gc\\_ResultMsg\(\)\*\*](#) (deprecated)
- [\*\*gc\\_CCLibIDToName\(\)\*\*](#)

## `gc_Extension( )`

**Name:** `int gc_Extension(target_type, target_id, ext_id, parmblkp, retblkp, mode)`

**Inputs:**

<code>int target_type</code>	• type of target object (line device or call reference number)
<code>long target_id</code>	• ID of target: either line device handle or call reference number
<code>unsigned char ext_id</code>	• extension function identifier as specified by the technology
<code>GC_PARM_BLK* parmblkp</code>	• pointer to <a href="#">GC_PARM_BLK</a> as specified by the technology
<code>GC_PARM_BLK* retblkp</code>	• pointer to address of <a href="#">GC_PARM_BLK</a> union where parameter values are to be returned. Optional depending on technology.
<code>unsigned long mode</code>	• async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`

**Category:** FTE

**Mode:** asynchronous or synchronous

**Platform and Technology:** Springware: T-1/E-1 (PDKRT only)<sup>†</sup>, ISDN<sup>†</sup>, Analog (PDKRT only)

DM3: ISDN<sup>†</sup>

SS7<sup>†</sup>

IP (host-based stack<sup>†</sup> and embedded stack<sup>†</sup>)

<sup>†</sup>See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The `gc_Extension( )` function provides a generic interface for technology-specific features, which otherwise require the support of the lower-level call control library APIs.

A target object is a configurable basic entity and is represented by its target type and target ID. For the `gc_Extension( )` function, the target type identifies whether the basic entity is either a line device (`GCTGT_GCLIB_CHAN`) or a call (`GCTGT_GCLIB_CRN`). The target ID identifies the specific target object (for example, line device handle or CRN), which is generated by Global Call at runtime. See [Section 6.2, “Target Objects”](#), on page 417 for more information.

The *Global Call Technology User's Guides* provide the details on how to map the Global Call extension function to the intended corresponding technology-specific function provided via the call control library. This mapping is done via the function identifier, which is used by the call control library as an index for the requested extension function, should more than one extension function be provided by the library.

The [EXTENSIONEVTBLK](#) extension block structure contains technology-specific information and is referenced via the `extevtdatap` pointer in the [METAEVENT](#) structure associated with the `GCEV_EXTENSION` and `GCEV_EXTENSIONCMPLT` events.

If the **gc\_Extension( )** function is used to transmit information to the remote end point, the application at the remote end point receives a GCEV\_EXTENSION event.

Parameter	Description
<b>target_type</b>	target object type. Valid types are: <ul style="list-style-type: none"> <li>• GCTGT_GCLIB_CHAN</li> <li>• GCTGT_GCLIB_CRN</li> </ul> See <a href="#">Table 22, “GC Parameter Entry List Maintained in GCLIB”</a> , on page 422 for details about these two types.
<b>target_id</b>	target object identifier. This identifier, along with <b>target_type</b> , uniquely specifies the target object. Valid identifiers are: <ul style="list-style-type: none"> <li>• line device handle</li> <li>• call reference number</li> </ul>
<b>ext_id</b>	extension function identifier, if multiple extension functions are provided by the technology. If multiple extension functions are not provided, this parameter is not used. See the appropriate <i>Global Call Technology User’s Guide</i> for actual definitions.
<b>parmbldp</b>	points to <a href="#">GC_PARM_BLK</a> as defined by the technology. This parameter is not used in cases where extension functions do not have any parameters.
<b>retbldp</b>	points to the <a href="#">GC_PARM_BLK</a> used for potential return of parameter values, as defined by the technology. This pointer is not used in cases where extension functions do not return one or more parameter values to the caller. See the appropriate <i>Global Call Technology User’s Guide</i> for definitions of parameters contained within the GC_PARM_BLK.  If a GC_PARM_BLK buffer is returned, the parameters in this buffer must be processed or copied prior to the next Global Call function call. The reason for this is that the GC_PARM_BLK buffer will be deallocated in a subsequent Global Call function call.
<b>mode</b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

## ■ Termination Events

GCEV\_EXTENSIONCMPLT

indicates the successful completion of the extension function.

GCEV\_TASKFAIL

indicates that the function failed. See the “Error Handling” section in the *Global Call API Programming Guide*.

## ■ Cautions

- The [EXTENSIONEVTBLK](#) extension block structure has a persistence only until the next call of [gc\\_GetMetaEvent\( \)](#) or [gc\\_GetMetaEventEx\( \)](#). In other words, any information contained or referenced in a GCEV\_EXTENSION or GCEV\_EXTENSIONCMPLT event

must be either processed or cached within the application, or risk being lost upon the next call of `gc_GetMetaEvent()` or `gc_GetMetaEventEx()`.

- The `GC_PARAM_BLK` buffer, if one is returned via the `retblkp` pointer, also has a guaranteed persistence only until the next Global Call function call. Any parameters within this buffer must be processed or copied within the application, or risk being lost when the next Global Call function call is invoked.

## ■ Errors

If this function returns `<0` to indicate failure, use the `gc_ErrorInfo()` function for error information. If the `GCEV_TASKFAIL` event is received, use the `gc_ResultInfo()` function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

```
#include "stdio.h"
#include "gclib.h"
#include "gcerr.h"
#include "gcisdn.h"

/* ISDN Example: Send an PROGRESS message to the network. */
/* Refer to "ISDN Technology User's Guide" for details. */
int ExtSendCallProgressMsg(CRN crn)
{
    GC_PARAM_BLK parmblkp = NULL; /* input parameter block pointer */
    GC_PARAM_BLK retblkp = NULL; /* pointer for output parameter block (unused) */
    int progval = 0; /* call progress indicator value */
    int retval = GC_SUCCESS;
    GC_INFO gc_error_info; /* GlobalCall error information data */

    printf("\n Enter call progress indicator value : ");
    progval = getchar();

    /* allocate GC_PARAM_BLK for call progress message parameter */
    gc_util_insert_parm_ref(&parmblkp, GCIS_SET_CALLPROGRESS,
                           GCIS_PARAM_CALLPROGRESS_INDICATOR, sizeof(int), &progval);

    if (parmblkp == NULL)
    {
        /* memory allocation error */
        return(-1);
    }

    /* transmit PROGRESS message to network */
    retval = gc_Extension(GCTGT_GCLIB_CRN, crn, GCIS_EXID_CALLPROGRESS, parmblkp,
                          &retblkp, EV_SYNC);

    if (retval != GC_SUCCESS)
    {
        gc_ErrorInfo(&gc_error_info);
        printf ("Error : gc_Extension() on CRN: 0x%x, GC ErrorValue: 0x%x - %s,\n",
                CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
                crn, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
}
```

```
/* free the parameter block */  
gc_util_delete_parm_blk (parmbkp);  
  
return (retval);  
}
```

■ **See Also**

None

## gc\_GetAlarmConfiguration()

**Name:** int gc\_GetAlarmConfiguration(linedev, aso\_id, alarm\_list, alarm\_config\_type)

**Inputs:** LINEDEV linedev                      • line device ID  
          unsigned long aso\_id                • alarm source object ID  
          ALARM\_LIST \*alarm\_list           • pointer to alarm list to be filled in  
          int alarm\_config\_type             • alarm information type

**Returns:** 0 if successful  
          <0 if failure

**Includes:** gclib.h  
          gcerr.h

**Category:** GCAMS

**Mode:** synchronous

**Platform and** Springware: T-1/E-1, ISDN

**Technology:** DM3: T-1/E-1, ISDN  
                  IP (host-based stack)

---

### ■ Description

The **gc\_GetAlarmConfiguration()** function retrieves alarm configuration parameter values. This function returns the configuration of all the alarms in the specified alarm source object (ASO) or for a specified ASO within a given line device.

This function generates a list of alarms and their associated values for a specified type of alarm information. The information included in the list is determined by the value specified for the **alarm\_config\_type** parameter. For example, if **ALARM\_CONFIG\_NOTIFY** is specified, the list will contain the current value of the notification attribute (**ALARM\_NOTIFY** or **ALARM\_NONOTIFY**) for all the alarms in the specified alarm source object.

The information for each alarm is returned in the **ALARM\_FIELD** data structure within the **ALARM\_LIST** data structure.

Parameter	Description
<b>linedev</b>	line device. If this value is 0, then the function returns the current default configuration of the alarm source object, which may have been modified using the <b>gc_SetAlarmConfiguration()</b> function.
<b>aso_id</b>	ASO ID. Use the <b>gc_AlarmSourceObjectNameToID()</b> function to obtain the ASO ID for the desired alarm source object.  ALARM_SOURCE_ID_NETWORK_ID can be used if the network ASO ID associated with the line device is desired. To obtain an <b>aso_id</b> not associated with a line device, NULL can be used.

Parameter	Description
<b>alarm_list</b>	<p>points to the alarm list to be filled in. The alarm list contains the list of alarms to be passed to the application by Global Call. The fields in the <a href="#">ALARM_LIST</a> structure provide the following information:</p> <ul style="list-style-type: none"> <li>• <b>n_alarms</b> – contains the number of alarms in the list</li> <li>• <b>alarm_fields</b> – an array of data structures that contain information about the list of alarms</li> </ul> <p>The following information is contained in the <a href="#">ALARM_FIELD</a> data structures for each alarm in the list:</p> <ul style="list-style-type: none"> <li>• <b>alarm_number</b> – contains the alarm number</li> <li>• <b>alarm_data</b> – contains the value of the information that was requested, for example, <b>ALARM_NOTIFY</b> or <b>ALARM_NONOTIFY</b> if <b>alarm_config_type</b> = <b>ALARM_CONFIG_NOTIFY</b>. The data type of <b>alarm_data</b> is dependent upon the <b>alarm_config_type</b> parameter. See Table 5 for data types.</li> </ul> <p>See <a href="#">ALARM_LIST</a>, on page 368 and <a href="#">ALARM_FIELD</a>, on page 367 for more information about those data structures.</p>
<b>alarm_config_type</b>	alarm configuration type. See Table 5 for the possible values. The information is returned in the data field in <b>alarm_list</b> for all alarms in the specified alarm source object.

Table 5. Alarm Configuration Types

Configuration Type	Retrievals	Data Type	Possible Values	Alarms Returned
ALARM_CONFIG_BLOCKING	Current value of the blocking attribute.	int	ALARM_BLOCKING = blocking alarm ALARM_NONBLOCKING = non-blocking alarm	All
ALARM_CONFIG_NOTIFY	Current value of the notification attribute	int	ALARM_NONOTIFY = do not notify when alarm received ALARM_NOTIFY = notify when alarm received	All
ALARM_CONFIG_NAME	Alarm name. The alarm_data field in the ALARM_LIST structure contains a pointer to the name for each alarm. The linedev parameter is ignored for ALARM_CONFIG_NAME	char*	The name of the alarm.	All



Table 5. Alarm Configuration Types (Continued)

Configuration Type	Retrievals	Data Type	Possible Values	Alarms Returned
ALARM_CONFIG_STATUS	Current alarm status (alarm on or alarm off)	int	ALARM_ON = the alarm is active ALARM_OFF = the alarm is inactive	All
ALARM_CONFIG_STATUS_BLOCKING	Current alarm status (alarm on or alarm off) for those alarms whose blocking/non-blocking attribute is set to blocking. This information is returned in the alarm_data field in <b>alarm_list</b> . (The alarm_data field is of type int.)	int	ALARM_ON = blocking alarm is active ALARM_OFF = blocking alarm is inactive	Only those alarms with blocking attribute set to on.

#### ■ Termination Events

None

#### ■ Cautions

When requesting the names of the alarms, do not overwrite the space pointed to by the data field as this points to private, internal Global Call space.

#### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

#### ■ Example

```
#include <stdio.h>
#include <stdlib.h>
#include <gclib.h>
#include <gcerr.h>
#include <dtilib.h>

LINEDEV    ldev;
ALARM_LIST alarm_names;
ALARM_LIST alarm_status_blocking;
ALARM_LIST alarm_blocking;
ALARM_LIST alarm_notify;
int        aso_id;
int        i;
int        DTE1_LOS_status;
GC_INFO    gc_error_info;    /* GlobalCall error information data */

aso_id = ALARM_SOURCE_ID_DM3_E1;    /* assume DM3 running E1 */
```

```

/*****
/* Demonstrate retrieving alarm names */
*****/

if (gc_GetAlarmConfiguration(ldev, aso_id, &alarm_names, ALARM_CONFIG_NAME) < 0)
{
    /* process error as shown */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error : gc_GetAlarmConfiguration() on device handle: 0x%lx,
            GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
            ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
            gc_error_info.ccLibId, gc_error_info.ccLibName,
            gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}
else
{
    printf("There are %d alarms for linedevice %d and they are:\n",
          alarm_names.n_alarms, ldev);
    for( i = 0; i < alarm_names.n_alarms; i++)
    {
        printf("alarm ID = %d\talarm name = %s\n",
              alarm_names.alarm_fields[i].alarm_number,
              alarm_names.alarm_fields[i].alarm_data.paddress);
    }
}

/*****
/* Demonstrate retrieving alarm status for blocking alarms */
*****/

if (gc_GetAlarmConfiguration(ldev, aso_id, &alarm_status_blocking,
                             ALARM_CONFIG_STATUS_BLOCKING) < 0)
{
    /* process error as shown */
    /* Note: gc_GetAlarmConfiguration() is a GC only function */
    /* hence the cclib error message is not printed */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error : gc_GetAlarmConfiguration() on device handle: 0x%lx,
            GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
            ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
            gc_error_info.ccLibId, gc_error_info.ccLibName,
            gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}
else
{
    /* print status of all blocking alarms */
    printf("Alarm status for blocking alarms for linedevice %d is:\n", ldev);
    for( i = 0; i < alarm_status_blocking.n_alarms; i++)
    {
        printf("alarm ID = %d\tAlarm status = %s\n",
              alarm_status_blocking.alarm_fields[i].alarm_number,
              alarm_status_blocking.alarm_fields[i].alarm_data.intvalue

              == ALARM_ON ? "On" : "Off");
    }
}
/* now check and see if DTE1_LOS is ON */
DTE1_LOS_status = ALARM_OFF;
for( i = 0; i < alarm_status_blocking.n_alarms; i++)
{
    if (alarm_names.alarm_fields[i].alarm_number == DTE1_LOS)
    {
        DTE1_LOS_status = alarm_names.alarm_fields[i].alarm_data.intvalue;
    }
}

```

```

        break;
    }
}
if (i < alarm_status_blocking.n_alarms)
{
    printf("Blocking alarm status of DTE1_LOS is %s\n",
           DTE1_LOS_status == ALARM_ON ? "On" : "Off");
}
else
{
    printf("Did not find DTE1_LOS in blocking alarms\n");
}

/*****
/* Demonstrate retrieving the current value of blocking/non-blocking */
*****/
if (gc_GetAlarmConfiguration(ldev, aso_id, &alarm_blocking,
                             ALARM_CONFIG_BLOCKING) < 0)
{
    /* process error as shown */
    /* Note: gc_GetAlarmConfiguration() is a GC only function */
    /* hence the cclib error message is not printed */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error : gc_GetAlarmConfiguration () on device handle: 0x%lx,
            GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
            ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
            gc_error_info.ccLibId, gc_error_info.ccLibName,
            gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}

printf("Blocking/non-blocking status for linedevice %d is:\n", ldev);
for( i = 0; i < alarm_status_blocking.n_alarms; i++)
{
    printf("alarm ID = %d\tBlocking status = %s\n",
           alarm_status_blocking.alarm_fields[i].alarm_number,
           alarm_status_blocking.alarm_fields[i].alarm_data.intvalue
           == ALARM_BLOCKING ? "Blocking" : "Non-Blocking");
}

/*****
/* Demonstrate retrieving the current value of notify/do-not notify */
*****/
if (gc_GetAlarmConfiguration(ldev, aso_id, &alarm_notify,
                             ALARM_CONFIG_NOTIFY) < 0)
{
    /* process error as shown */
    /* Note: gc_GetAlarmConfiguration() is a GC only function */
    /* hence the cclib error message is not printed */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error : gc_GetAlarmConfiguration () on device handle: 0x%lx,
            GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
            ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
            gc_error_info.ccLibId, gc_error_info.ccLibName,
            gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}

printf("Notify/Do not notify status for linedevice %d is:\n", ldev);
for( i = 0; i < alarm_notify.n_alarms; i++)
{
    printf("alarm ID = %d\tNotify status = %s\n",
           alarm_notify.alarm_fields[i].alarm_number,
           alarm_notify.alarm_fields[i].alarm_data.intvalue == ALARM_NOTIFY
           ? "Notify" : "Do not notify");
}

```

■ **See Also**

- [gc\\_SetAlarmConfiguration\( \)](#)

## `gc_GetAlarmFlow()`

**Name:** `int gc_GetAlarmFlow(aso_id, flow)`

**Inputs:** `unsigned long aso_id`      • alarm source object ID  
`int *flow`      • pointer to where current flow value is to be stored

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** GCAMS

**Mode:** synchronous

**Platform and** Springware: T-1/E-1, ISDN

**Technology:** DM3: T-1/E-1, ISDN  
IP (host-based stack)

---

### ■ Description

The **`gc_GetAlarmFlow()`** function indicates which alarms are sent to the application. This function frees the application from having to store the current alarm flow setting.

The alarm flow defines which alarms are sent to the application. Alarm flow is controlled on a line device basis using the **`gc_SetAlarmFlow()`** function.

Parameter	Description
<b>aso_id</b>	alarm source object (ASO) ID. Use the <a href="#">gc_AlarmSourceObjectNameToID()</a> function to obtain the ASO ID for the desired alarm source object.  ALARM_SOURCE_ID_NETWORK_ID can be used if the network ASO ID associated with the line device is desired. To obtain an <b>aso_id</b> not associated with a line device, NULL can be used.
<b>flow</b>	points to the destination for the current flow control value. Possible values for flow control are: <ul style="list-style-type: none"> <li>• ALARM_FLOW_ALWAYS – all alarms with notification on are sent to the application (default)</li> <li>• ALARM_FLOW_ALWAYS_BLOCKING – only blocking alarms with notification on are sent to the application. Non-blocking alarms are not sent.</li> <li>• ALARM_FLOW_FIRST_AND_LAST – only the first alarm on and the last alarm off are sent to the application (if their notification attribute is on). Both blocking and non-blocking alarms are eligible to be sent.</li> <li>• ALARM_FLOW_FIRST_AND_LAST_BLOCKING – only the first blocking alarm on and the last blocking alarm off are sent to the application (if their notification attribute is on)</li> </ul>

#### ■ Termination Events

None

#### ■ Cautions

None

#### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

#### ■ Example

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

int          flow;
int          rc;
GC_INFO     gc_error_info;    /* GlobalCall error information data */

/* This code assumes that the application is running over DM3 E1 */
rc = gc_GetAlarmFlow(ALARM_SOURCE_ID_DM3_E1, &flow);
```

```

if (rc < 0)
{
    /* process error as shown */
    /* Note: gc_GetAlarmFlow() is a GC only function */
    /* hence the cclib error message is not printed */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error : gc_GetAlarmFlow() on aso_id: 0x%lx, GC ErrorValue: 0x%hx - %s,
            CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
            ALARM_SOURCE_ID_DM3_E1, gc_error_info.gcValue, gc_error_info.gcMsg,
            gc_error_info.ccLibId, gc_error_info.ccLibName,
            gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}
else switch(flow)
{
    case ALARM_FLOW_ALWAYS:
        printf("Current alarm flow is ALARM_FLOW_ALWAYS\n");
        break;

    case ALARM_FLOW_ALWAYS_BLOCKING:
        printf("Current alarm flow is ALARM_FLOW_ALWAYS_BLOCKING\n");
        break;

    case ALARM_FLOW_FIRST_AND_LAST:
        printf("Current alarm flow is ALARM_FLOW_FIRST_AND_LAST\n");
        break;

    case ALARM_FLOW_FIRST_AND_LAST_BLOCKING:
        printf("Current alarm flow is ALARM_FLOW_FIRST_AND_LAST_BLOCKING\n");
        break;

    default:
        printf("Unknown alarm flow, value = %d\n", flow);
        break;
}

```

## ■ See Also

- [`gc\_SetAlarmFlow\(\)`](#)

## gc\_GetAlarmParm( )

**Name:** int gc\_GetAlarmParm(linedev, aso\_id, ParmSetID, alarm\_parm\_list, mode)

**Inputs:**

LINEDEV linedev	• line device
unsigned long aso_id	• alarm source object ID
int ParmSetID	• parameter set ID
ALARM_PARM_LIST *alarm_parm_list	• pointer to alarm parameter list
unsigned long mode	• sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** GCAMS

**Mode:** synchronous

**Platform and** Springware: T-1/E-1, ISDN

**Technology:** IP (host-based stack† and embedded stack†)  
†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The **gc\_GetAlarmParm( )** function retrieves parameter data for a parameter set ID for one or more (alarm, parameter ID) pairs. The alarm number and parameter set ID pairs are specified in ALARM\_PARM\_FIELD in the **alarm\_parm\_list** list.

The parameter set ID (**ParmSetID**) is the ID of a set of associated parameters as defined by the alarm source object (ASO). An example of one parameter set is the parameters that can be set by the **dt\_setparm( )** function. Another example of a parameter set is the parameters that are set by the **dt\_setalarm( )** function. Both functions are in the Springware DT library.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>aso_id</b>	alarm source object (ASO) ID. Use the <a href="#">gc_AlarmSourceObjectNameToID( )</a> function to obtain the ASO ID for the desired alarm source object.  ALARM_SOURCE_ID_NETWORK_ID can be used if the network ASO ID is not known.
<b>ParmSetID</b>	ID of the parameter set as defined by the ASO. See the <i>Global Call API Programming Guide</i> for a list of the Global Call-aware ASO IDs and the appropriate <i>Global Call Technology User's Guide</i> for technology-specific ASO IDs.



Parameter	Description
<code>alarm_parm_list</code>	points to the alarm parameter list. See <a href="#">ALARM_PARM_LIST</a> , on page 370 for more information.
<code>mode</code>	set to <code>EV_SYNC</code> for synchronous execution (only synchronous mode is supported)

## ■ Termination Events

None

## ■ Cautions

- The parameters are alarm source object dependent. Detailed knowledge of the alarm source object is necessary in order to use these parameters properly.
- Not all alarm source objects support the retrieval of all user-set parameters.
- There must be sufficient space available for the information to be retrieved.
- If a failure occurs during a call to the `gc_GetAlarmParm()` function, the information in `aso_list` is considered invalid.
- The application must initialize the pointer in the `alarm_data` field to ensure that it points to valid user data space.
- Alarm source objects will return an int through the `pstruct` field of the `GC_PARM` structure instead of the `intvalue` field of the structure. The `pstruct` pointer must be initialized to point to where the int is to be stored.
- Additional header files for other libraries will most likely be required. See the appropriate *Global Call Technology User's Guide* for more information.

## ■ Errors

If this function returns `<0` to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

```
#include <stdio.h>
#include <stdlib.h>
#include <gcclib.h>
#include <gcerr.h>
#include <dtilib.h>                                /* for ASO symbols */

int          rc;
LINEDEV      linedev;
ALARM_PARM_LIST alarm_parm_list;
GC_INFO      gc_error_info; /* GlobalCall error information data */

/*
-- This code assumes that linedev is already assigned
-- it also assumes that linedev's alarm source object
-- is known to be Springware T1 this could have been
-- done via gc_GetAlarmSourceObjectNetworkID
*/
```

```
/* init all to 0 */
memset(&alarm_parm_list, '\0', sizeof(ALARM_PARM_LIST));

/* get 1 parameter */
alarm_parm_list.n_parms = 1;

/* get # of out of frame errors to allow before sending an alarm */
alarm_parm_list.alarm_parm_fields[0].alarm_parm_number.intvalue = DTG_OOFMAX;

rc = gc_GetAlarmParm(linedev, ALARM_SOURCE_ID_SPRINGWARE_T1, ParmSetID_parm,
                    &alarm_parm_list, EV_SYNC);
if (rc < 0)
{
    /* get and process the error */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_GetAlarmParm() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
           CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
           linedev, gc_error_info.gcValue, gc_error_info.gcMsg,
           gc_error_info.ccLibId, gc_error_info.ccLibName,
           gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}
else
{
    printf("for linedev %d, # of out of frame errors to allow before sending \
    an alarm is %d\n", linedev,
    alarm_parm_list.alarm_parm_fields[0].alarm_parm_data.intvalue);
}
```

#### ■ See Also

- [gc\\_SetAlarmParm\(\)](#)

## `gc_GetAlarmSourceObjectList()`

**Name:** `int gc_GetAlarmSourceObjectList(linedev, ByNameOrById, aso_list)`

**Inputs:**

<code>LINEDEV linedev</code>	• line device
<code>int ByNameOrById</code>	• specifies whether list is a list of ASO names or a list of ASO IDs
<code>ALARM_SOURCE_OBJECT_LIST *aso_list</code>	• pointer to alarm source object list

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** GCAMS

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN  
DM3: T-1/E-1, ISDN  
IP (host-based stack)

### ■ Description

The `gc_GetAlarmSourceObjectList()` function retrieves a list of all alarm source objects associated with a specified line device. The list can contain either alarm source object (ASO) names or ASO IDs.

If the call control library associated with the specified line device uses the Global Call Alarm Management System (GCAMS), then the first entry in the list of ASOs retrieved by `gc_GetAlarmSourceObjectList()` is the network ASO ID.

Parameter	Description
<b>linedev</b>	line device
<b>ByNameOrById</b>	specifies whether the function is to return a list of ASO names or a list of ASO IDs. Valid values are: <ul style="list-style-type: none"> <li>• <code>ASO_LIST_BY_NAME</code></li> <li>• <code>ASO_LIST_BY_ID</code></li> </ul>
<b>aso_list</b>	points to the alarm source object list. The Global Call API fills in this data structure with a list of all alarm source objects and their names for the requested line device. NULL is not allowed. The <code>aso_data</code> field of <a href="#">ALARM_SOURCE_OBJECT_FIELD</a> is of type <code>char *</code> if the request is “by name”. If the request is by ID, the field is typed an unsigned long. See <a href="#">ALARM_SOURCE_OBJECT_LIST</a> , on page 372 for a description of the data structure.

## ■ Termination Events

None

## ■ Cautions

None

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

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

LINEDEV          linedev;
ALARM_SOURCE_OBJECT_LIST  aso_list_by_name;
ALARM_SOURCE_OBJECT_LIST  aso_list_by_id;
int               rc, i;
GC_INFO          gc_error_info; /* GlobalCall error information data */

/*
-- This code assumes that linedev is already assigned
*/
rc = gc_GetAlarmSourceObjectList(linedev, ASO_LIST_BY_NAME, &aso_list_by_name);
if (rc < 0)
{
    /* get and process the error */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_GetAlarmSourceObjectList() on device handle: 0x%x,
GC ErrorValue: 0x%x - %s, CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
linedev, gc_error_info.gcValue, gc_error_info.gcMsg,
gc_error_info.ccLibId, gc_error_info.ccLibName,
gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}
else
{
    rc = gc_GetAlarmSourceObjectList(linedev, ASO_LIST_BY_ID, &aso_list_by_id);
    if (rc < 0)
    {
        /* get and process the error */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetAlarmSourceObjectList() on device handle: 0x%x,
GC ErrorValue: 0x%x - %s, CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
linedev, gc_error_info.gcValue, gc_error_info.gcMsg,
gc_error_info.ccLibId, gc_error_info.ccLibName,
gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    else
    {
        /*

```

```
-- This code assumes that aso_list_by_id.n_asos == aso_list_by_name.n_asos
*/
printf("Alarm source objects associated with linedev %d are: \n", linedev);
for (i = 0; i < aso_list_by_id.n_asos; i++)
{
    printf("\tID: %d\t\tName: %s\n",
        aso_list_by_id.aso_fields[i].aso_data.intvalue,
        aso_list_by_name.aso_fields[i].aso_data.paddress);
}
}
```

#### ■ See Also

None

## **gc\_GetAlarmSourceObjectNetworkID( )**

**Name:** int gc\_GetAlarmSourceObjectNetworkID(linedev, aso\_networkID)

**Inputs:** LINEDEV linedev • line device  
unsigned long \*aso\_networkID • pointer to where network ASO ID is to be stored

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** GCAMS

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN

DM3: T-1/E-1, ISDN  
IP (host-based stack)

---

### ■ Description

The **gc\_GetAlarmSourceObjectNetworkID( )** function retrieves the ID of the layer 1 alarm source object (ASO) associated with a specified line device. The network alarm source object ID represents the network handle for the source of physical layer alarms.

Parameter	Description
<b>linedev</b>	line device
<b>aso_networkID</b>	points to where the alarm source network ID is stored

### ■ Termination Events

None

### ■ Cautions

None

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

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

void gc_demo_open_ElorTl_channel(int index)
{
    char        str[MAX_STRING_SIZE];
    int         ASOnetworkID;
    GC_INFO     gc_error_info;    /* GlobalCall error information data */

    printandlog(index, MISC, NULL, "E1 or T1 device being opened");

    /* perform port data structure initialization */

    if (gc_OpenEx(&port[index].ldev, port[index].devname, EV_SYNC, (void *)&port[index])
        != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_OpenEx() on devname: %s, GC ErrorValue: 0x%xh - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%xh - %s\n",
                port[index].devname, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    sprintf(str, "gc_OpenEx(devicename=%s, mode=EV_SYNC) Success", port[index].devname);

    printandlog(index, GC_APICALL, NULL, str);

    /* a real application might wish to cache ASOnetworkID in the port data structure */
    if (gc_GetAlarmSourceObjectNetworkID(port[index].ldev, &ASOnetworkID) != GC_SUCCESS)
    {
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetAlarmSourceObjectNetworkID() on device handle: 0x%xh,
                GC ErrorValue: 0x%xh - %s, CCLibID: %i - %s, CC ErrorValue: 0x%xh - %s\n",
                port[index].ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    else
    {
    }

    printf("networkID for linedev %d is %d\n", port[index].ldev, ASOnetworkID);
}
```

## ■ See Also

None

## gc\_GetANI( )

**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 call handling (deprecated)

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN†, Analog†

DM3: T-1/E-1, ISDN†, Analog†  
SS7  
IP (embedded stack†)

†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

**Note:** The `gc_GetANI( )` function is deprecated in this software release. The suggested equivalent is `gc_GetCallInfo( )`.

The `gc_GetANI( )` function retrieves automatic number identification (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. The call must be in the Offered state to retrieve the ANI information.

See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>crn</b>	call reference number
<b>ani_buf</b>	address of the buffer where ANI is to be loaded. The returned digits will be terminated with “\0”.

### ■ Termination Events

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\_ADDR\_SIZE.



- If this function is invoked for an unsupported technology, the function fails. The error value `EGC_UNSUPPORTED` will be the Global Call value returned when the [gc\\_ErrorInfo\(\)](#) function is used to retrieve the error code.

## ■ Errors

If this function returns `<0` to indicate failure, use the [gc\\_ErrorInfo\(\)](#) function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```

#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)
 * 4. The event is determined to be a GCEV_OFFERED event
 */
int get_ani(void)
{
    CRN          crn;          /* call reference number */
    char          ani_buf[GC_ADDR_SIZE]; /* Buffer for ANI digits */
    GC_INFO      gc_error_info; /* GlobalCall error information data */

    /*
     * Get the calling party number
     */
    crn = metaevent.crn;
    if (gc_GetANI(crn, ani_buf) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo(&gc_error_info);
        printf ("Error: gc_GetAPI() on crn: 0x%lx, GC ErrorValue: 0x%hx - %s,\n",
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                crn, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

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

```

## ■ See Also

- [gc\\_GetCallInfo\(\)](#)
- [gc\\_ReqANI\(\)](#)
- [gc\\_WaitCall\(\)](#)

## **gc\_GetBilling( )**

**Name:** int gc\_GetBilling(crn, billing\_buf)

**Inputs:** CRN crn • call reference number  
char \*billing\_buf • pointer to buffer for storing billing information

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** optional call handling

**Mode:** synchronous

**Platform and Technology:** Springware: ISDN†

†See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

The **gc\_GetBilling( )** function retrieves the billing information associated with the specified call. The charge information is in ASCII string format. The information is retrieved from the Global Call software.

The **gc\_GetBilling( )** function is used only in ISDN applications that use AT&T's Vari-A-Bill service. See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>crn</b>	call reference number
<b>billing_buf</b>	address of the buffer where the requested information is stored. See the appropriate <i>Global Call Technology User's Guide</i> for the format and usage of this field.

### ■ Termination Events

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 Global Call value returned when the **gc\_ErrorInfo( )** function is used to retrieve the error code.

## ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ 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) 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 <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>

/* The gc_GetBilling() function is exclusively used for the AT&T Vari-A-Bill
   service. */

int get_billing_info(CRN crn, char *billing_buffer)
{
    LINEDEV ldev;          /* Line device */
    GC_INFO gc_error_info; /* GlobalCall error information data */

    if(gc_CRN2LineDev(crn, &ldev) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_CRN2LineDev() on crn: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                crn, gc_error_info.gcValue, gc_error_info.gCMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    if(gc_GetBilling(crn, billing_buffer) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetBilling() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                ldev, gc_error_info.gcValue, gc_error_info.gCMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    return(0);
}
```

## ■ See Also

None

**gc\_GetCallInfo( )****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  
 gccrr.h

**Category:** interface specific**Mode:** synchronous**Platform and** All†**Technology:** †See the *Global Call Technology User's Guides* for additional information.■ **Description**

The **gc\_GetCallInfo( )** function retrieves information associated with the call. You can use this function at any time. The application can retrieve only one type of information at a time. The type of information that can be retrieved depends on the technology and whether the call is inbound or outbound (see Table 6).

See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>crn</b>	call reference number
<b>info_id</b>	ID of the information to be collected. See Table 6.
<b>valuep</b>	buffer address where the requested information is stored

**Table 6. gc\_GetCallInfo( ) info\_id Parameter ID Definitions**

info_id Parameter	Definition	Technology	valuep Format
CALLINFOTYPE	Call related information: CHARGE or NO CHARGE for the current call.	E-1 CAS†	string
CALLNAME	Calling party's name	Analog	string
CALLTIME	Time and date call was made	Analog	string
† Not supported in all protocols; check your protocol in the <i>Global Call Country Dependent Parameters (CDP) Reference</i> or contact your Intel® Dialogic sales representative. ‡ The <b>gc_GetCallInfo( )</b> function with an <b>info_id</b> parameter of CONNECT_TYPE should only be called after a GCEV_CONNECTED or GCEV_MEDIADETECTED event is received.			

**Table 6. *gc\_GetCallInfo()* info\_id Parameter ID Definitions (Continued)**

info_id Parameter	Definition	Technology	valuep Format
CATEGORY_DIGIT	Category digit. Determined by the protocol/technology.	E-1 CAS (Springware only), SS7	character
CONNECT_TYPE‡	<p>Defines the type of connection returned by call progress analysis. When <b>gc_GetCallInfo()</b> is used to retrieve information for an outbound call, one of the following is returned:</p> <ul style="list-style-type: none"> <li>GCCT_DISCARDED – the call was disconnected by the remote end or the network and media type detection was discarded</li> <li>GCCT_CAD – connection due to cadence break</li> <li>GCCT_FAX – connection due to fax machine detection</li> <li>GCCT_FAX1 – connection due to CNG fax tone detection (PDKRT only)</li> <li>GCCT_FAX2 – connection due to CED fax tone detection (PDKRT only)</li> <li>GCCT_INPROGRESS – media detection information is not yet available</li> <li>GCCT_LPC – connection due to change in loop current</li> <li>GCCT_PVD – connection due to positive voice detection</li> <li>GCCT_NA – media type detection was not enabled</li> <li>GCCT_PAMD – connection due to positive answering machine detection</li> <li>GCCT_UNKNOWN – unknown media type answered</li> </ul>	Analog, E-1 CAS, T-1 Robbed Bit†, R4 on DM3	character
DESTINATION_ADDRESS	Request the called party address (for example, DNIS).	E-1 CAS, T-1 Robbed Bit, Analog, ISDN PRI, R4 on DM3 (T-1 and ISDN), SS7†, IP	digits
ORIGINATION_ADDRESS	Request the calling party address (for example, ANI).	E-1 CAS, T-1 Robbed Bit, Analog, ISDN PRI, R4 on DM3 (T-1 and ISDN), SS7†, IP	digits
<p>† Not supported in all protocols; check your protocol in the <i>Global Call Country Dependent Parameters (CDP) Reference</i> or contact your Intel® Dialogic sales representative.</p> <p>‡ The <b>gc_GetCallInfo()</b> function with an <b>info_id</b> parameter of CONNECT_TYPE should only be called after a GCEV_CONNECTED or GCEV_MEDIADETECTED event is received.</p>			

Table 6. gc\_GetCallInfo( ) info\_id Parameter ID Definitions (Continued)

info_id Parameter	Definition	Technology	valuep Format
PRESENT_RESTRICT	The calling party presentation restriction: <ul style="list-style-type: none"> <li>• 0 – presentation allowed</li> <li>• 1 – presentation restricted</li> <li>• 2 – address not available (national use only)</li> </ul>	SS7	byte
U_IES	Unformatted user-to-user Information Elements (IEs) retrieved in CCITT format. The application needs to allocate sufficient memory (up to 256 bytes) to hold the retrieved IEs. The IEs are returned as raw data and must be parsed and interpreted by the application. Use the GC_IE_BLK data structure to retrieve unprocessed IEs. See also the <i>Global Call ISDN Technology User's Guide</i> for DPNSS-related IEs.	ISDN	string
UUI	User-to-User Information. See also the <i>Global Call ISDN Technology User's Guide</i> for technology-specific information.	ISDN	string
† Not supported in all protocols; check your protocol in the <i>Global Call Country Dependent Parameters (CDP) Reference</i> or contact your Intel® Dialogic sales representative. ‡ The gc_GetCallInfo( ) function with an info_id parameter of CONNECT_TYPE should only be called after a GCEV_CONNECTED or GCEV_MEDIADETECTED event is received.			

## ■ Termination Events

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. For ANI and DNIS, the largest expected string length 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 Global Call value returned when the [gc\\_ErrorInfo\( \)](#) function is used to retrieve the error code.

## ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ 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) has been
 *    called to convert the event into metaevent.
 * 4. a GCEV_OFFERED has been detected.
 */

#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <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 ldev;          /* Line device */
    GC_INFO gc_error_info; /* GlobalCall error information data */

    if(gc_CRN2LineDev(crn, &ldev) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_CRN2LineDev() on crn: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                crn, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    if(gc_GetCallInfo(crn, info_id, valuep) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetCallInfo() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    return(0);
}
```

■ **See Also**

None



## `gc_GetCallProgressParm()`

**Name:** `int gc_GetCallProgressParm(linedev, parm)`

**Inputs:** `LINEDEV linedev`      • Global Call line device handle  
`DX_CAP *parm`      • pointer to the device information structure

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** system controls and tools

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1 (PDKRT only)

---

### ■ Description

The **`gc_GetCallProgressParm()`** function retrieves protocol call progress parameters. The parameters are contained in the `DX_CAP` data.

Parameter	Description
<b><code>linedev</code></b>	Global Call line device handle
<b><code>parm</code></b>	points to the <code>DX_CAP</code> call progress information structure. See <a href="#">DX_CAP</a> , on page 375 for more information.

### ■ Termination Events

None

### ■ Cautions

None

### ■ Errors

If this function returns <0 to indicate failure, use the **`gc_ErrorInfo()`** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

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

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

/*
 * Data structure which stores all information for each line
 */
static struct linebag {
    LINEDEV    ldev;          /* GlobalCall API line device handle */
    CRN        crn;           /* GlobalCall API call handle */
    int        blocked;       /* channel blocked/unblocked */
    int        networkh;      /* network handle */
} port[MAX_CHAN+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 and voice handle is stored in linebag structure "port"
 */
int call_getcallprogressparm(int port_num, DX_CAP *infop)
{
    GC_INFO    gc_error_info; /* GlobalCall error information data */
    /* Find info for this time slot, specified by 'port_num' */
    /* (assumes port_num is valid) */
    pline = port + port_num;

    /* Get information about the device and store it in the location pointed
     * to by infop */
    if (gc_GetCallProgressParm(pline->ldev, infop) == -1)
    {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetCallProgressParm() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                pline->ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    return (0);
}
```

## ■ See Also

- [gc\\_SetCallProgressParm\(\)](#)

## `gc_GetCallState()`

**Name:** `int gc_GetCallState(crn, state_ptr)`

**Inputs:** `CRN crn` • call reference number  
`int *state_ptr` • pointer to variable for returning call state

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** system controls and tools

**Mode:** synchronous

**Platform and** All

**Technology:**

---

### ■ Description

The `gc_GetCallState()` function retrieves the state of the call associated with the call reference number (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. State transition diagrams and call state definitions are presented in the *Global Call API Programming Guide*.

Parameter	Description
<code>crn</code>	call reference number
<code>state_ptr</code>	<p>points to the location where the call state value will be returned. Possible state values are:</p> <ul style="list-style-type: none"><li>• <code>GCST_ACCEPTED</code> – An inbound call was accepted; the call is in the Accepted state.</li><li>• <code>GCST_ALERTING</code> – The call is waiting for the destination party to answer; the call is in the Alerting state (alerted sent or received).</li><li>• <code>GCST_CALLROUTING</code> – Exists for an incoming call when the user has sent an acknowledgment that all call information necessary to effect call establishment has been received.</li></ul> <p><b>Note:</b> <code>GCST_CALLROUTING</code> is not supported on DM3 boards.</p> <ul style="list-style-type: none"><li>• <code>GCST_CONNECTED</code> – An inbound or outbound call was connected; the call is in the Connected state.</li><li>• <code>GCST_DETECTED</code> – An incoming call has been received but has not yet been offered to the application; the call is in the Detected state.</li><li>• <code>GCST_DIALING</code> – An outbound call request was received; the call is in the Dialing state.</li></ul>

Parameter	Description
	<ul style="list-style-type: none"> <li>GCST_DIALTONE – The line device is ready to make a consultation call in a supervised transfer; the call is in the Dialtone state. (Supported only under the PDKRT call control library.)</li> <li>GCST_DISCONNECTED – The call was disconnected from the network, the call is in the Disconnected state.</li> <li>GCST_GETMOREINFO – Exists for an inbound call when the network has received an acknowledgment of the call establishment request, which permits the network to send additional call information (if any) in the overlap mode.</li> <li>GCST_IDLE – The call is not active; the call is in the Idle state.</li> <li>GCST_NULL – The call was released; the call is in the Null state.</li> <li>GCST_OFFERED – An inbound call was received, the call is in the Offered state.</li> <li>GCST_ONHOLD – The call was placed on hold; the call is in the OnHold state.</li> <li>GCST_ONHOLDPENDINGTRANSFER – The call is on hold and waiting to be transferred to another call (supervised transfer); the call is in the OnHoldPendingTransfer state. (Supported only under the PDKRT call control library.)</li> <li>GCST_PROCEEDING – Exists for an outbound call when the user has received an acknowledgment that all call information necessary to effect call establishment has been received and the call is proceeding.</li> </ul> <p><b>Note:</b> GCST_PROCEEDING is not supported on DM3 boards.</p> <ul style="list-style-type: none"> <li>GCST_SENDMOREINFO – Exists for an outbound call when the user has received an acknowledgment of the call establishment request that permits or requests the user to send additional call information to the network in overlap mode.</li> </ul>

## ■ Termination Events

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.
- The GCST\_CALLROUTING and GCST\_PROCEEDING call states are not supported on DM3 boards.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gccerr.h* file. If the error returned is technology

specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

### ■ Example

```
#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 */
    GC_INFO gc_error_info; /* GlobalCall error information data */
    /* 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_CRN2LineDev( crn, &ldev) != GC_SUCCESS)
        {
            /* get and process error */
            gc_ErrorInfo( &gc_error_info );
            printf ("Error: gc_CRN2LineDev() on crn: 0x%lx, GC ErrorValue: 0x%hx - %s,
                    CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                    crn, gc_error_info.gcValue, gc_error_info.gcMsg,
                    gc_error_info.ccLibId, gc_error_info.ccLibName,
                    gc_error_info.ccValue, gc_error_info.ccMsg);
            return (gc_error_info.gcValue);
        }

        if (gc_GetCallState( crn, &call_state) != GC_SUCCESS)
        {
            /* process error return as shown */
            gc_ErrorInfo( &gc_error_info );
            printf ("Error: gc_GetCallState() on devive handle: 0x%lx,
                    GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                    ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                    gc_error_info.ccLibId, gc_error_info.ccLibName,
                    gc_error_info.ccValue, gc_error_info.ccMsg);
            return (gc_error_info.gcValue);
        }
    }

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

*gc\_GetCallState( ) — retrieve the state of the call*



■ **See Also**

None

## `gc_GetConfigData()`

**Name:** `int gc_GetConfigData(target_type, target_id, target_datap, time_out, request_idp, mode)`

**Inputs:**

<code>int target_type</code>	• target object type
<code>long target_id</code>	• target object ID
<code>GC_PARM_BLK target_datap</code>	• pointer to the data from target object
<code>int time_out</code>	• time-out in seconds
<code>long * request_idp</code>	• pointer to the location of retrieve request ID
<code>unsigned long mode</code>	• async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gcerr.h`

**Category:** RTCM

**Mode:** asynchronous or synchronous

**Platform and Technology:** Springware: T-1/E-1 (PDKRT only), ISDN†, Analog (PDKRT only)

†See the *Global Call Technology User's Guides* for additional information

### ■ Description

The `gc_GetConfigData()` function supports the Real Time Configuration Management (RTCM) feature. The `gc_GetConfigData()` function retrieves parameter values for a given target object. The `gc_GetConfigData()` function also outputs a unique request ID, which is used by the application to trace the function call. All subsequent references to this request must be made using the request ID.

Configuration data for multiple parameters can be retrieved in a single call to the `gc_GetConfigData()` function. However, only one target object can be accessed in a single function call.

See [Table 22, “GC Parameter Entry List Maintained in GCLIB”](#), on page 422 and [Table 23, “Examples of Parameter Entry List Maintained in CCLIB”](#), on page 424. For more information about the RTCM feature, see the *Global Call API Programming Guide*.

Parameter	Description
<code>target_type</code>	target object type
<code>target_id</code>	target object identifier. This identifier along with <code>target_type</code> , uniquely specifies the target object.
<code>target_datap</code>	specifies the pointer to the <a href="#">GC_PARM_BLK</a> structure. The structure contains the parameter configuration data to be retrieved.

Parameter	Description
<b>time_out</b>	time interval (in seconds) during which the data of the target object must be retrieved; if the interval is exceeded, the request to retrieve is ignored. This parameter is ignored when set to 0.  The time-out option is supported in synchronous mode only. If the data requested is not retrieved within the time specified, the <b>gc_GetConfigData( )</b> function will return EGC_TIMEOUT.
<b>request_idp</b>	points to the retrieve request ID, which is generated by Global Call
<b>mode</b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

The parameter **target\_datap** points to the location of the GC\_PARM\_BLK that stores the configuration parameter data to be retrieved. Memory allocation and deallocation of the GC\_PARM\_BLK data block is done by the Global Call utility functions (**gc\_util\_xxx**). See [Section 1.12, “GC\\_PARM\\_BLK Utility Functions”](#), on page 18 for more information.

It is the Global Call application’s responsibility to use the Global Call utility functions to allocate an appropriately sized data block memory (GC\_PARM\_BLK) for the configuration parameters and to insert parameter information (such as the set ID, parm ID, value buffer size, and value buffer) into the GC\_PARM\_BLK data block. After calling the **gc\_GetConfigData( )** function, the successfully retrieved parameter value(s) will be passed back to the GC\_PARM\_BLK (value buffer fields). After the Global Call application finishes using the GC\_PARM\_BLK, the application should deallocate the GC\_PARM\_BLK data block using the **gc\_util\_delete\_parm\_blk( )** function. See [GC\\_PARM\\_BLK](#), on page 392 for more information.

The termination events, GCEV\_GETCONFIGDATA and GCEV\_GETCONFIGDATA\_FAIL, both have an associated [GC\\_RTCM\\_EVTDATA](#) data structure (to which the evtdatap field in [METAEVENT](#) points) that includes the request ID, additional message, retrieved [GC\\_PARM\\_BLK](#), etc. See [GC\\_RTCM\\_EVTDATA](#), on page 396 for more information.

When the **gc\_GetConfigData( )** function is called to retrieve the configuration data of a group of parameters, the request will terminate if the retrieval of any single parameter fails. Use the **gc\_ErrorInfo( )** function (for function return values) or **gc\_ResultInfo( )** function (for termination events) to find out what kind of error occurred and which parameter data (set ID and parm ID) failed to be retrieved. See the “Error Handling” section in the *Global Call API Programming Guide* for more information about these error functions.

## ■ Termination Events

### GCEV\_GETCONFIGDATA

indicates that the **gc\_GetConfigData( )** function was successful, that is, the configuration data has been retrieved successfully.

### GCEV\_GETCONFIGDATA\_FAIL

indicates that the **gc\_GetConfigData( )** function failed, that is, an error occurred during retrieval.



## ■ Cautions

Only synchronous mode is supported for the following target objects: GCTGT\_GCLIB\_SYSTEM, GCTGT\_CCLIB\_SYSTEM, GCTGT\_PROTOCOL\_SYSTEM, and GCTGT\_FIRMWARE\_NETIF. Otherwise, the function will return the async mode error.

## ■ Errors

If this function returns <0 to indicate failure, use the [gc\\_ErrorInfo\(\)](#) function for error information. If the GCEV\_GETCONFIGDATA\_FAIL event is received, use the [gc\\_ResultInfo\(\)](#) function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

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

#define MAXCHAN 30
#define T_MAX_NAMESTRING 50

/*
 * Data structure which stores all information about each line device
 */
struct linebag
{
    LINEDEV    ldev;          /* Lline device ID */
    CRN        crn;          /* CRN */
    int        call_sate;
} port[MAXCHAN];

struct linebag    *pline;
int                get_parm_valuetype(GC_PARM_DATA * parmdatap);
void              print_parm_data(int valuetype, GC_PARM_DATAP parmdatap);

/* get the configuration of a GC target object */
int get_configuration(int target_type, int port_num, GC_PARM_BLK * parm_blkp,
                    long * request_idp)
{
    int    result;
    long    target_id = 0;
    GC_INFO t_gcinfo;

    switch (target_type)
    {
        case GCTGT_GCLIB_SYSTEM:
            target_id = GC_LIB;
            break;
        case GCTGT_CCLIB_SYSTEM:
            /*get cclib ID */
            break;
        case GCTGT_PROTOCOL_SYSTEM:
            /*get protocol ID by calling gc_QueryConfigData() with protocol name*/
            break;
        case GCTGT_GCLIB_CRN:
    }
```

```

        case GCTGT_CCLIB_CRN:
            /* If the target object is a CRN */
            pline = port + port_num;
            target_id = pline->crn;
            break;

        case GCTGT_GCLIB_NETIF:
        case GCTGT_CCLIB_NETIF:
        case GCTGT_GCLIB_CHAN:
        case GCTGT_CCLIB_CHAN:
        case GCTGT_PROTOCOL_CHAN:
        case GCTGT_FIRMWARE_CHAN:
            /* If the target object is a line device (time slot or network interface) */
            pline = port + port_num;
            target_id = pline->ldev;
            break;

        default:
            /* Otherwise: return -1 */
            printf("Unsupported target type: 0x%lx\n", target_type);
            return -1;
            break;
    }
    /* Call gc_GetConfigData() function */
    result = gc_GetConfigData(target_type, target_id, parm_blkp, 0, request_idp,
                             EV_SYNC);
    if (result != GC_SUCCESS)
    {
        /* retrieve error values by calling gc_ErrorInfo */
        if (gc_ErrorInfo(&t_gcinfo) == GC_SUCCESS)
        {
            printf("Error on target type: 0x%lx, target ID: 0x%lx\n", target_type,
                  target_id);
            printf("with GC Error 0x%xh: %s\n", t_gcinfo.gcValue, t_gcinfo.gcMsg);
            printf("CCLib %d(%s) Error - 0x%xh: %s\n", t_gcinfo.ccLibId,
                  t_gcinfo.ccLibName, t_gcinfo.ccValue, t_gcinfo.ccMsg);
            printf("Additional message: %s\n", t_gcinfo.additionalInfo);
        }
        else
        {
            printf("gc_ErrorInfo() failure");
        }
    }
    return result;
}

/* retrieve the individual parameters from GC_PARM_BLK */
void retrieve_parm_data_blk(GC_PARM_BLK * parm_data_blkp)
{
    /* get the first parm from the block */
    GC_PARM_DATA * parmdatap = gc_util_next_parm(parm_data_blkp, NULL);
    int valuetype = 0;
    while (parmdatap != NULL)
    {
        /* get the type of parm value */
        valuetype = get_parm_valuetype(parmdatap);
        if (valuetype != -1)
        {
            /* if the value type is valid, then print the parm data */
            print_parm_data(valuetype, parmdatap);
        }
        /* get the next parm from the block */
        parmdatap = gc_util_next_parm(parm_data_blkp, parmdatap);
    }
}

```

```

int main()
{
    int    port_num = 0;
    char    t_StrValue[T_MAX_NAMESTRING] = "";
    long    request_id = 0;

    /* To call the GC PARM utility function to insert a parameter,
       the pointer to GC_PARM_BLK must be initialized to NULL */
    GC_PARM_BLK * parm_data_blkp = NULL;

    /* First call GC PARM utility function to insert the parameters to be retrieved */
    /* 1. insert device name parm */
    gc_util_insert_parm_ref(&parm_data_blkp, GCSET_DEVICEINFO, GCPARM_DEVICENAME,
        T_MAX_NAMESTRING, t_StrValue);
    /* 2. insert network handle parm */
    gc_util_insert_parm_val(&parm_data_blkp, GCSET_DEVICEINFO, GCPARM_NETWORKH,
        sizeof(int), 0);
    /* 3. insert voice name parm */
    gc_util_insert_parm_ref(&parm_data_blkp, GCSET_DEVICEINFO, GCPARM_VOICENAME,
        T_MAX_NAMESTRING, t_StrValue);
    /* 4. insert voice handle parm */
    gc_util_insert_parm_val(&parm_data_blkp, GCSET_DEVICEINFO, GCPARM_VOICEH,
        sizeof(int), 0);
    /* 5. insert board line device ID parm */
    gc_util_insert_parm_val(&parm_data_blkp, GCSET_DEVICEINFO, GCPARM_BOARD_LDID,
        sizeof(long), 0);
    /* 6. insert protocol ID parm */
    gc_util_insert_parm_val(&parm_data_blkp, GCSET_PROTOCOL, GCPARM_PROTOCOL_ID,
        sizeof(long), 0);
    /* 7. insert protocol name parm */
    gc_util_insert_parm_ref(&parm_data_blkp, GCSET_PROTOCOL, GCPARM_PROTOCOL_NAME,
        T_MAX_NAMESTRING, t_StrValue);
    /* 8. insert call event parm */
    gc_util_insert_parm_val(&parm_data_blkp, GCSET_CALLEVENT_MSK, GCPARM_GET_MSK,
        sizeof(long), 0);
    /* 9. insert call state parm */
    gc_util_insert_parm_val(&parm_data_blkp, GCSET_CALLSTATE_MSK, GCPARM_GET_MSK,
        sizeof(long), 0);

    /* after creating the GC_PARM_BLK, call get_configuration function to get the
       configuration data block */
    get_configuration(GCTGT_GCLIB_CHAN, port_num, parm_data_blkp, &request_id);
    /* retrieve individual parameters from the GC_PARM_BLK */
    retrieve_parm_data_blk(parm_data_blkp);
    /* delete the parm data block after using */
    gc_util_delete_parm_blk(parm_data_blkp);
    return 0;
}

/* get the type of parm value for a given parm */
int get_parm_valuetype(GC_PARM_DATA * parmdatap)
{
    switch (parmdatap->set_ID)
    {
        case GCSET_DEVICEINFO:
        {
            switch (parmdatap->parm_ID)
            {
                case GCPARM_DEVICENAME:
                case GCPARM_VOICENAME:
                    /* a string value */
                    return GC_VALUE_STRING;
                break;
            }
        }
    }
}

```

```
        case GCPARM_NETWORKH:
        case GCPARM_VOICEH:
        case GCPARM_CALLSTATE:
            /* an integer value */
            return GC_VALUE_INT;
            break;

        case GCPARM_BOARD_LDID:
            /* a long value */
            return GC_VALUE_LONG;
            break;

        default:
            return -1;
            break;
    }
    break;
}
case GCSET_PROTOCOL:
{
    switch (parmdatap->parm_ID)
    {
        case GCPARM_PROTOCOL_ID:
            /* a long value */
            return GC_VALUE_LONG;
            break;
        case GCPARM_PROTOCOL_NAME:
            /* a string value */
            return GC_VALUE_STRING;
            break;
    }
    break;
}

case GCSET_CALLEVENT_MSK:
case GCSET_CALLSTATE_MSK:
case GCSET_CRN_INDEX:
    /* a long value */
    return GC_VALUE_LONG;
    break;

case GCSET_CCLIB_INFO:
{
    switch (parmdatap->parm_ID)
    {
        case GCPARM_CCLIB_NAME:
            /* a string value */
            return GC_VALUE_STRING;
            break;
        case GCPARM_CCLIB_ID:
            /* an integer value */
            return GC_VALUE_INT;
            break;
        default:
            return -1;
            break;
    }
    break;
}

default:
    return -1;
    break;
}
return -1;
}
```

```

/* print the parm data */
void print_parm_data(int valuetype, GC_PARM_DATAP parmdatap)
{
    int      t_IntVal;
    char      t_StrVal[T_MAX_NAMESTRING];
    short     t_ShortVal;
    long      t_LongVal;
    char      t_CharVal;

    switch (valuetype)
    {
        case GC_VALUE_INT:
            if (parmdatap->value_size == sizeof(int))
            {
                memcpy(&t_IntVal, parmdatap->value_buf, sizeof(int));
                printf("Retrieve Set ID = %d and Parm ID = %d with value = %d \n",
                    parmdatap->set_ID, parmdatap->parm_ID, t_IntVal);
            }
            break;
        case GC_VALUE_LONG:
            if (parmdatap->value_size == sizeof(long))
            {
                memcpy(&t_LongVal, parmdatap->value_buf, sizeof(long));
                printf("Retrieve Set ID = %d and Parm ID = %d with value = %d \n",
                    parmdatap->set_ID, parmdatap->parm_ID, t_LongVal);
            }
            break;
        case GC_VALUE_SHORT:
            if (parmdatap->value_size == sizeof(short))
            {
                memcpy(&t_ShortVal, parmdatap->value_buf, sizeof(short));
                printf("Retrieve Set ID = %d and Parm ID = %d with value = %d \n",
                    parmdatap->set_ID, parmdatap->parm_ID, t_ShortVal);
            }
            break;
        case GC_VALUE_CHAR:
            if (parmdatap->value_size == sizeof(char))
            {
                memcpy(&t_CharVal, parmdatap->value_buf, sizeof(char));
                printf("Retrieve Set ID = %d and Parm ID = %d with value = %c \n",
                    parmdatap->set_ID, parmdatap->parm_ID, t_CharVal);
            }
            break;
        case GC_VALUE_STRING:
            if (parmdatap->value_size == T_MAX_NAMESTRING)
            {
                strcpy(t_StrVal, (const char*)parmdatap->value_buf);
                printf("Retrieve Set ID = %d and Parm ID = %d with value = %s \n",
                    parmdatap->set_ID, parmdatap->parm_ID, t_StrVal);
            }
            break;
        default:
            printf("Retrieve Unsupported parm Set ID = %d and Parm ID = %d\n",
                parmdatap->set_ID, parmdatap->parm_ID);
            break;
    }
}

```

#### ■ See Also

- [gc\\_SetConfigData\(\)](#)
- [gc\\_QueryConfigData\(\)](#)

## gc\_GetCRN( )

**Name:** int gc\_GetCRN(crn\_ptr, metaeventp)

**Inputs:** CRN \*crn\_ptr • 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 controls and tools

**Mode:** synchronous

**Platform and Technology:** All

---

### ■ Description

The **gc\_GetCRN( )** function retrieves a call reference number for the event to which the pointer **metaeventp** is pointing. This **metaeventp** pointer is filled in by the **gc\_GetMetaEvent( )** function or, for the Windows extended asynchronous model only, the **gc\_GetMetaEventEx( )** function.

The **gc\_GetCRN( )** function is supported for backward compatibility only. The application can access the CRN directly from the crn field in the **METAEVENT** structure pointed to by the **metaeventp** parameter retrieved by the **gc\_GetMetaEvent( )** or **gc\_GetMetaEventEx( )** (Windows only) function.

If the event is associated with a call, the **crn\_ptr** parameter contains a pointer to the CRN. If the event is associated with a line device, the **crn\_ptr** parameter is 0.

**Note:** The application can access the line device directly from the linedev field in the **METAEVENT** structure pointed to by the **metaeventp** parameter retrieved by the **gc\_GetMetaEvent( )** or **gc\_GetMetaEventEx( )** (Windows only) function.

Parameter	Description
<b>crn_ptr</b>	points to the memory address where the call reference number is stored
<b>metaeventp</b>	points to the <b>METAEVENT</b> structure filled in by the <b>gc_GetMetaEvent( )</b> or <b>gc_GetMetaEventEx( )</b> function. See <b>METAEVENT</b> , on page 403 for more information.

### ■ Termination Events

None

## ■ Cautions

None

## ■ Errors

If this function returns <0 to indicate failure, use the **`gc_ErrorInfo()`** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
#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)
 */
CRN get_crn(METAEVENT *metaeventp)
{
    CRN          crn;          /* call reference number */
    GC_INFO      gc_error_info; /* GlobalCall error information data */

    if (gc_GetCRN(&crn, metaeventp) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo(&gc_error_info);
        printf ("Error: gc_GetCRN() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                metaeventp->evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    else {
        /*
         * Else return the CRN and next issue the GlobalCall function call
         * using the CRN.
         */
        return(crn);
    }
}
```

## ■ See Also

- [gc\\_GetLineDev\(\)](#)
- [gc\\_GetMetaEvent\(\)](#)
- [gc\\_GetMetaEventEx\(\)](#) (Windows extended asynchronous model only)
- [gc\\_MakeCall\(\)](#)
- [gc\\_WaitCall\(\)](#)

## gc\_GetCTInfo( )

**Name:** int gc\_GetCTInfo(linedev, ct\_devinfo)

**Inputs:** LINEDEV linedev • Global Call line device  
CT\_DEVINFO \*ct\_devinfo • pointer to device information structure

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** system controls and tools

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1 (PDKRT only)

DM3: T-1/E-1, ISDN, Analog  
IP (embedded stack)

### ■ Description

The **gc\_GetCTInfo( )** function retrieves CT Bus time slot information. The information includes the device family, device mode, type of network interface, bus architecture, PCM encoding, etc. The information is returned in the **CT\_DEVINFO** structure. The valid values for each member of the **CT\_DEVINFO** structure are defined in *gclib.h*.

Parameter	Description
<b>linedev</b>	Global Call device handle
<b>ct_devinfo</b>	points to the <b>CT_DEVINFO</b> device information structure. See <b>CT_DEVINFO</b> , on page 374 for more information.

### ■ Termination Events

None

### ■ Cautions

None

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).



## ■ Example

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

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

/*
 * Data structure which stores all information for each line
 */

static struct linebag {
    LINEDEV    ldev;        /* GlobalCall API line device handle */
    CRN        crn;         /* GlobalCall API call handle */
    int        blocked;     /* channel blocked/unblocked */
    int        networkh;    /* network handle */
} port[MAX_CHAN+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 and voice handle is stored in linebag structure "port"
 */

int call_getctinfo(int port_num, CT_DEVINFO *infop)
{
    GC_INFO     gc_error_info; /* GlobalCall error information data */

    /* Find info for this time slot, specified by 'port_num' */
    /* (assumes port_num is valid) */

    pline = port + port_num;

    /* Get information about the device and store it in the location pointed
       to by infop */

    if (gc_GetCTInfo(pline->ldev, infop) == -1)
    {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetCTInfo() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                pline->ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    return (0);
}
```

## ■ See Also

- [gc\\_GetXmitSlot\(\)](#)
- [gc\\_Listen\(\)](#)
- [gc\\_UnListen\(\)](#)

## gc\_GetDNIS( )

**Name:** int gc\_GetDNIS(crn, dnis)

**Inputs:** CRN crn • call reference number  
char \*dnis • pointer to buffer to store DNIS info

**Returns:** 0 if successful  
<0 if failure

**Includes:** gcilib.h  
gcerr.h

**Category:** optional call handling (deprecated)

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN†

DM3: T-1/E-1, ISDN†, Analog  
SS7†

IP (embedded stack†)

†See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

**Note:** The `gc_GetDNIS( )` function is deprecated in this software release. The suggested equivalent is `gc_GetCallInfo( )`.

The `gc_GetDNIS( )` function retrieves the dialed number identification service (DNIS) information (DDI digits) associated with a specific call reference number (CRN). The DDI digits are in ASCII string format and end with “\0”. See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>crn</b>	call reference number
<b>dnis</b>	points to the buffer where the DNIS is stored

### ■ Termination Events

None

### ■ Cautions

- The buffer pointed to by **dnis** 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 Global Call value returned when the `gc_ErrorInfo()` function is used to retrieve the error code.

## ■ Errors

If this function returns `<0` to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

```
#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)
 * 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 */
    GC_INFO      gc_error_info;      /* GlobalCall error information data */

    /* 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_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetDNIS() on crn: 0x%lx, GC ErrorValue: 0x%hx - %s,\n",
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                crn, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    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);
}
```

■ **See Also**

- [gc\\_CallAck\(\)](#)
- [gc\\_GetANI\(\)](#) (deprecated)
- [gc\\_GetCallInfo\(\)](#)
- [gc\\_WaitCall\(\)](#)

## `gc_GetFrame()`

**Name:** `int gc_GetFrame(linedev, l2_blkp)`

**Inputs:** `LINEDEV linedev` • line device handle for D channel  
`GC_L2_BLK *l2_blkp` • pointer to the buffer to store received frame info

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** interface specific

**Mode:** synchronous

**Platform and** Springware: ISDN

**Technology:** DM3: ISDN

---

### ■ Description

**Note:** The `gc_GetFrame()` function is deprecated in this software release. The suggested equivalent is `gc_Extension()`.

The `gc_GetFrame()` function retrieves a Layer 2 frame received by the application. This function is used after a `GCEV_L2FRAME` event is received. Each `GCEV_L2FRAME` event is associated with one frame. This function is used for the data link layer only.

**Note:** To enable Layer 2 access, set parameter number 24 to 01 in the firmware parameter file. When Layer 2 access is enabled, only the `gc_GetFrame()` and `gc_SndFrame()` functions can be used (no calls can be made).

Parameter	Description
<code>linedev</code>	Global Call line device handle for the D channel
<code>l2_blkp</code>	points to the memory location where the received frame information is to be stored

The retrieved frame uses the `GC_L2_BLK` data structure; see `GC_L2_BLK`, on page 389. Also, see the example code for details.

### ■ Termination Events

None

### ■ Cautions

The `gc_GetFrame()` function is called only after a `GCEV_L2FRAME` event is received. See also the appropriate *Global Call Technology User's Guide* for technology-specific information and to the protocol-specific parameter file.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <gcLib.h>
#include <gcerr.h>
#include <gcisdn.h>

METAEVENT metaevent;

int send_frame(GC_L2_BLK *sndfrmptr)
{
    LINEDEV    ldev;          /* Line device */
    GC_INFO    gc_error_info; /* GlobalCall error information data */
    char       devname[] = ":N_dtiB1:P_isdn";

    /*
     * Do the following:
     *   1. Open the D channel
     *   2. Send the frame
     */

    if(gc_OpenEx(&ldev, devname, EV_SYNC, &ldev) < 0) {
        /* Process the error as shown in the Example
         * code of the gc_ErrorInfo() function description */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_OpenEx() on devname: %s, GC ErrorValue: 0x%x - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
                devname, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    if(gc_SndFrame(ldev, sndfrmptr) != GC_SUCCESS) {
        /* Process the error as shown in the Example
         * code of the gc_ErrorInfo() function description */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetFrame() on linedev: 0x%x, GC ErrorValue: 0x%x - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
                ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    return 0;
}

/* Retrieve events from SRL */
int evt_hdlr()
{
    LINEDEV    ldev;          /* Line device */
    GC_L2_BLK  rcvfrmptr;     /* Buffer to store received frame */
    int        retcode;        /* Return value */
    GC_INFO    gc_error_info; /* GlobalCall error information data */
}
```

```
retcode = gc_GetMetaEvent(&metaevent);

if (retcode < 0 ) {
    /* get and process the error */
} else {
    /* Continue with normal processing */
}

ldev = metaevent.evtdev;

switch(metaevent.evtttype) {

case GCEV_L2FRAME:
    /* retrieve signaling information from queue */
    if ( gc_GetFrame(ldev, &recvfrmptr) != GC_SUCCESS) {
        /* Process the error as shown in the Example
         * code of the gc_ErrorInfo() function description */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetFrame() on linedev: 0x%x, GC ErrorValue: 0x%x - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
                ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    else {
        /* succeeded, process signaling information */
    }
    break;
}
return 0;
}
```

#### ■ See Also

- [gc\\_Extension\(\)](#)
- [gc\\_SetUserInfo\(\)](#)
- [gc\\_SndFrame\(\)](#) (deprecated)

## gc\_GetInfoElem( )

**Name:** int gc\_GetInfoElem(linedev, iep)

**Inputs:** LINEDEV linedev • line device handle of the B channel board  
GC\_IE\_BLK \*iep • pointer to the IE buffer

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** interface specific (deprecated)

**Mode:** synchronous

**Platform and Technology:** Springware: ISDN

### ■ Description

**Note:** The **gc\_GetInfoElem( )** function is deprecated in this software release. The suggested equivalent is **gc\_GetUserInfo( )**. For DM3 boards, **gc\_GetSigInfo( )** can be used as an alternative.

The **gc\_GetInfoElem( )** function retrieves information elements (IEs) associated with a line device for an incoming message. The **gc\_GetInfoElem( )** function must be used immediately after the message is received if the application requires the call information. The call control library will not queue the call information; subsequent messages on the same line device will be discarded if the previous messages are not retrieved.

Parameter	Description
<b>linedev</b>	B channel board line device handle
<b>iep</b>	points to the starting address of the IE block that is stored in the <a href="#">GC_IE_BLK</a> structure; see <a href="#">GC_IE_BLK</a> , on page 387. Also, see the Example code for details.

### ■ Termination Events

None

### ■ Cautions

None

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology



specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

### ■ Example

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

#define MAX_QUEUE_SIZE    20

METAEVENT metaevent;

int evt_hdlr()
{
    LINEDEV    ldev;          /* Line device */
    CRN        crn;           /* Call Reference Number */
    GC_IE_BLK  ie_blk;        /* Information Element */
    GC_INFO    gc_error_info; /* GlobalCall error information data */
    int        retcode;        /* Return Code */

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

    crn = metaevent.crn;
    if(gc_CRN2LineDev(crn, &ldev) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_CRN2LineDev() on crn: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                crn, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    switch(metaevent.evtttype) {

    case GCEV_NOTIFY:
        /* retrieve signaling information from queue */
        if (gc_GetInfoElem(ldev, &ie_blk) != GC_SUCCESS) {
            /* Process the error as shown in the Example
             * code of the gc_ErrorInfo() function description */
            gc_ErrorInfo( &gc_error_info );
            printf ("Error: gc_GetInfoElem() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
                    CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                    ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                    gc_error_info.ccLibId, gc_error_info.ccLibName,
                    gc_error_info.ccValue, gc_error_info.ccMsg);
            return (gc_error_info.gcValue);
        }
        else {
            /* succeeded, process signaling information */
        }
        break;
    }
    return 0;
}
```

■ See Also

- [gc\\_GetCallInfo\(\)](#)
- [gc\\_SetUserInfo\(\)](#)
- [gc\\_SetInfoElem\(\)](#) (deprecated)

## gc\_GetLineDev()

**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 controls and tools

**Mode:** synchronous

**Platform and** All

**Technology:**

---

### ■ Description

The **gc\_GetLineDev()** function retrieves a line device associated with an event received from the event queue. If this function is called for an event that is not a Global Call 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, for the Windows extended asynchronous model only, the **gc\_GetMetaEventEx()** function.

Parameter	Description
<b>linedevp</b>	points to the location where the output LINEDEV is stored
<b>metaeventp</b>	points to the <b>METAEVENT</b> structure filled in by the <b>gc_GetMetaEvent()</b> or the <b>gc_GetMetaEventEx()</b> function

### ■ Termination Events

None

### ■ Cautions

None

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*.

All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

### ■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <gcplib.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)
 * 4. The event is determined to be a GCEV_OFFERED event
 */
int get_linedev(LINEDEV *ldevp)
{
    GC_INFO          gc_error_info;    /* GlobalCall error information data */

    /*
     * Get Line Device corresponding to this event
     */
    if (gc_GetLineDev(ldevp, &metaevent) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetInfoElem() on linedev: 0x%x, GC ErrorValue: 0x%x - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
                metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

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

### ■ See Also

- [gc\\_GetCRN\(\)](#)
- [gc\\_GetMetaEvent\(\)](#)
- [gc\\_GetMetaEventEx\(\)](#) (Windows extended asynchronous model only)

## gc\_GetLinedevState( )

**Name:** int gc\_GetLinedevState(linedev, type, state\_buf)

**Inputs:** LINEDEV linedev • line device  
 int type • specifies type of line device  
 int \*state\_buf • pointer to location of line device state status

**Returns:** 0 if successful  
 <0 if failure

**Includes:** gclib.h  
 gcerr.h

**Category:** optional call handling

**Mode:** synchronous

**Platform and Technology:** Springware: ISDN  
 DM3: T-1/E-1, ISDN, Analog  
 SS7  
 IP (embedded stack)

### ■ Description

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

Parameter	Description
<b>linedev</b>	Global Call line device
<b>type</b>	specifies B channel or D channel device type associated with <b>linedev</b> . Valid values are: <ul style="list-style-type: none"> <li>GCGLS_BCHANNEL – get state of B channel</li> <li>GCGLS_DCHANNEL – get state of D channel</li> </ul>
<b>state_buf</b>	points to the location where the state information is to be stored. <p>Valid state values for the B channel are:</p> <ul style="list-style-type: none"> <li>GCLS_INSERVICE (0) – B channel is in service</li> <li>GCLS_MAINTENANCE (1) – B channel is in maintenance</li> <li>GCLS_OUT_OF_SERVICE (2) – B channel is out of service</li> </ul> <p>Valid state values for the D channel are:</p> <ul style="list-style-type: none"> <li>DATA_LINK_UP (0x02) – layer 2 operable</li> <li>DATA_LINK_DOWN (0x01) – layer 2 inoperable</li> </ul>

### ■ Termination Events

None

## ■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value `EGC_UNSUPPORTED` will be the Global Call value returned when the [gc\\_ErrorInfo\(\)](#) function is used to retrieve the error code.

## ■ Errors

If this function returns `<0` to indicate failure, use the [gc\\_ErrorInfo\(\)](#) function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <gcclib.h>
#include <gcerr.h>
#include <gcisdn.h>

int get_line_dev_state(void)
{
    LINEDEV  bdd;          /* Board level device */
    LINEDEV  ldev;         /* 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 */
    GC_INFO  gc_error_info; /* GlobalCall error information data */

    sprintf(bdevname, ":N_dtiB1:P_isdn");
    sprintf(ldevname, ":N_dtiB1T1:P_isdn");

    /*
     * Open two devices, dtiB1 and dtiB1T1.
     */
    if(gc_OpenEx(&bdd, bdevname, EV_SYNC, &bdd) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_OpenEx() on devname: %s, GC ErrorValue: 0x%x - %s,\n",
                CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
                bdevname, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    if(gc_OpenEx(&ldev, ldevname, EV_SYNC, &ldev) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_OpenEx() on devname: %s, GC ErrorValue: 0x%x - %s,\n",
                CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
                ldevname, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
}
```

```

/*
 * 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_ErrorInfo( &gc_error_info );
    printf ("Error: gc_GetLinedevState() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
           CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
           bdd, gc_error_info.gcValue, gc_error_info.gcMsg,
           gc_error_info.ccLibId, gc_error_info.ccLibName,
           gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}
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(ldev, type, &statebuf) != GC_SUCCESS) {
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_GetLinedevState() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
           CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
           ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
           gc_error_info.ccLibId, gc_error_info.ccLibName,
           gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}
else {
    printf("B Channel Status: %s\n", statebuf);
}

gc_Close(bdd);
gc_Close(ldev);

return (0);
}

```

## ■ See Also

- [gc\\_SetChanState\(\)](#)

## gc\_GetMetaEvent( )

**Name:** int gc\_GetMetaEvent(metaeventp)

**Inputs:** METAEVENT \*metaeventp • pointer to METAEVENT data structure

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** system controls and tools

**Mode:** synchronous

**Platform and** All

**Technology:**

---

### ■ Description

The **gc\_GetMetaEvent( )** function retrieves event information for the current SRL event that stores the Global Call and non-Global Call event information. This METAEVENT is a data structure that explicitly contains the information describing the SRL event to be returned to the Linux or Windows application. This data structure provides uniform information retrieval among call control libraries and across operating systems. See [METAEVENT](#), on page 403 for more information.

You must call the **gc\_GetMetaEvent( )** function to retrieve any Global Call event information and any other event information if you are not sure of the event type. If the metaevent is a Global Call event, the GCME\_GC\_EVENT bit in the METAEVENT flags field will be set. The Global Call-related fields of the METAEVENT data structure contain valid data **only** when the GCME\_GC\_EVENT bit is set. Do **not** use these fields if the bit is not set.

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 Global Call event or any other SRL event, the SRL event information (for example, evtdatap, evttype) may be retrieved from the METAEVENT data structure instead of using SRL functions to retrieve this information.

Parameter	Description
metaeventp	points to the <a href="#">METAEVENT</a> structure filled by this function

### ■ Termination Events

None



## ■ Cautions

- The `gc_GetMetaEvent()` function **must** be the first function called before processing any Global Call event.
- For Windows applications, when using the extended asynchronous model, the `gc_GetMetaEventEx()` function must be the first function called before processing any Global Call event. For all other Windows models, use the `gc_GetMetaEvent()` function.
- An application should call the `gc_GetMetaEvt()` function only once for a given event. Calling the function more than once will result in data corruption or an access violation.
- The event must be processed entirely in the same thread or all information about the event must be retrieved before processing the event in another thread.
- The `gc_GetMetaEvent()` and `gc_GetMetaEventEx()` functions should not be used in the same application.

## ■ Errors

If this function returns `<0` to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ 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 */
    }
}

```

The following code illustrates retrieving event information from the [METAEVENT](#) structure while making a call:

```

#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)
{
    GC_INFO gc_error_info; /* GlobalCall error information data */
    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 */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetMetaEvent() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    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 <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)
{
    GC_INFO gc_error_info;       /* GlobalCall error information data */

    long evtttype;               /* 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 */
            gc_ErrorInfo(&gc_error_info);
            printf ("Error: gc_GetMetaEvent() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
                    CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                    metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
                    gc_error_info.ccLibId, gc_error_info.ccLibName,
                    gc_error_info.ccValue, gc_error_info.ccMsg);
            return (gc_error_info.gcValue);
        }

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

```

```

        default:
            /* handle other events here */
            break;
    }
    break;

case OFFERED_STATE:
    switch (evtttype) {
        case GCEV_ACCEPT:
            pline -> state = ACCEPTED_STATE;
            answer_call();
            break;
        default:
            /* handle other events here */
            break;
    }
    break;
case ACCEPTED_STATE:
    switch (evtttype) {
        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 */
}
}
}

```

#### ■ See Also

- [gc\\_GetLineDev\(\)](#)
- [gc\\_GetMetaEventEx\(\)](#) (Windows extended asynchronous model only)
- [gc\\_ResultInfo\(\)](#)
- [gc\\_ResultValue\(\)](#) (deprecated)

## gc\_GetMetaEventEx( )

**Name:** int gc\_GetMetaEventEx(metaeventp, evt\_handle)  
(Windows extended asynchronous model only)

**Inputs:** METAEVENT \*metaeventp • pointer to METAEVENT data structure  
unsigned long evt\_handle • SRL event handle

**Returns:** 0 if successful  
<0 if failure

**Includes:** gcilib.h  
gcerr.h

**Category:** system controls and tools

**Mode:** synchronous

**Platform and** All

**Technology:**

### ■ Description

The **gc\_GetMetaEventEx()** function retrieves event information for an SRL event. The SRL event handle is specified in the **evt\_handle** parameter.

The **gc\_GetMetaEventEx()** function is used only in Windows applications that are running multithreads in the extended asynchronous model. The **gc\_GetMetaEventEx()** function returns a metaevent, which contains the SRL event information of the specified event handle, to the application. If your application uses the **sr\_WaitevtEx()** function, you must use **gc\_GetMetaEventEx()** to retrieve the metaevent.

The **gc\_GetMetaEventEx()** function has the same functionality and requirements as the **gc\_GetMetaEvent()** function. For more information, see the **gc\_GetMetaEvent()** function description.

Parameter	Description
<b>metaeventp</b>	points to the METAEVENT structure to be filled in by this function. See <a href="#">METAEVENT</a> , on page 403 for more information.
<b>evt_handle</b>	SRL event handle used to identify a particular event

### ■ Termination Events

None

### ■ Cautions

- The **gc\_GetMetaEvent()** or **gc\_GetMetaEventEx()** function **must** be the first function called before processing any Global Call event.

- When using the extended asynchronous model, the `gc_GetMetaEventEx()` function must be the first function called before processing any Global Call event. For all other Windows programming models, use the `gc_GetMetaEvent()` function.
- An application should call the `gc_GetMetaEventEx()` function only once for a given event. Calling the function more than once will result in data corruption or an access violation.
- The event must be processed entirely in the same thread or all information about the event must be retrieved before processing the event in another thread.
- 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.

## ■ Errors

If this function returns `<0` to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ 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;
hdlcnt[ hdlcnt++ ] = GetGlobalCallHandle();
hdlcnt[ hdlcnt++ ] = GetVoiceHandle();

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

rc = sr_waitevtEx( hdlcnt,
                  hdlcnt,
                  PollTimeout_ms,
                  &evtHdl
                );
if (rc != SR_TMOUT) {
    /*
     * Update
     */
    rc = gc_GetMetaEventEx(&g_Metaevent, evtHdl);
    if (rc != GC_SUCCESS) {
        /* process error return */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetMetaEventEx() on linedev: 0x%lx,
                GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
}
```

```
rc = vProcessCallEvents( );  
}
```

■ **See Also**

- [gc\\_GetCRN\(\)](#)
- [gc\\_GetLineDev\(\)](#)
- [gc\\_GetMetaEvent\(\)](#)
- [gc\\_ResultInfo\(\)](#)
- [gc\\_ResultValue\(\)](#) (deprecated)



## `gc_GetNetCRV()`

**Name:** `int gc_GetNetCRV(crn, netcrvp)`

**Inputs:** `CRN crn` • call reference number  
`int *netcrvp` • pointer to network call reference buffer

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gcLib.h`  
`gcErr.h`

**Category:** interface specific

**Mode:** synchronous

**Platform and Technology:** Springware: ISDN

DM3: ISDN†

†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

**Note:** The `gc_GetNetCRV()` function is deprecated in this software release. The suggested equivalent is `gc_Extension()`.

The `gc_GetNetCRV()` function retrieves the network call reference value (CRV) for a specified call reference number (CRN). The CRN is assigned during a `gc_MakeCall()` or `gc_WaitCall()` function call. If an invalid host CRN value is passed, for example, the CRN of an inactive call, the `gc_GetNetCRV()` function will return a value < 0 indicating failure.

Parameter	Description
<b>crn</b>	call reference number
<b>netcrvp</b>	points to memory location where the network call reference value (CRV) is to be stored

### ■ Termination Events

None

### ■ Cautions

None

### ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcErr.h` file. If the error returned is technology

specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

### ■ 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) has been
 *    called to convert the event into metaevent.
 * 4. a GCEV_OFFERED event has been detected.
 */

#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>

METAEVENT metaevent;

int get_net_crv()
{
    LINEDEV  ldev;          /* Line device */
    CRN      crn;           /* call reference number */
    int      netcrv;        /* Network Call Reference Value */
    GC_INFO  gc_error_info; /* GlobalCall error information data */

    /*
     * Do the following:
     * 1. Get the CRN from the metaevent
     * 2. Get the corresponding line device
     * 3. Proceed to retrieve the Call Reference Value as shown below
     */

    crn = metaevent.crn;

    if(gc_CRN2LineDev(crn, &ldev) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_CRN2LineDev() on crn: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                crn, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    if(gc_GetNetCRV(crn, &netcrv) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetNetCRV() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    return(0);
}
```



*retrieve the network call reference value — `gc_GetNetCRV()`*

■ **See Also**

- [gc\\_WaitCall\(\)](#)

## gc\_GetNetworkH( )

**Name:** int gc\_GetNetworkH(linedev, networkhp)

**Inputs:** LINEDEV linedev      • Global Call 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 controls and tools

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN

DM3: T-1/E-1, ISDN, Analog  
SS7†  
IP (embedded stack)

†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

**Note:** For technologies that support the [gc\\_GetResourceH\( \)](#) function, it is recommended that the [gc\\_GetResourceH\( \)](#) function be used instead of the [gc\\_GetNetworkH\( \)](#) function to retrieve the network device handle. See the [gc\\_GetResourceH\( \)](#) function description for more information.

The [gc\\_GetNetworkH\( \)](#) function retrieves the network device handle associated with the specified **linedev** line device. 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\( \)](#).

See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>networkhp</b>	points to the address where the network device handle is to be stored

### ■ Termination Events

None

### ■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value EGC\_UNSUPPORTED will be the Global Call value returned when the [gc\\_ErrorInfo\( \)](#) function is used to retrieve the error code.

## ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

```
#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)
{
    GC_INFO      gc_error_info;    /* GlobalCall error information data */
    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_ErrorInfo(&gc_error_info);
        printf("Error: gc_GetNetworkH() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,\n",
               CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
               ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
               gc_error_info.ccLibId, gc_error_info.ccLibName,
               gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    /*
     * Application may now generate or wait for a call on this line
     * device.
     */
    return (0);
}
```

## ■ See Also

- [gc\\_GetResourceH\(\)](#)
- [gc\\_GetVoiceH\(\)](#) (deprecated)

## gc\_GetParm( )

**Name:** int gc\_GetParm(linedev, parm\_id, valuep)

**Inputs:**

LINEDEV linedev	• Global Call 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

**Category:** system controls and tools

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1†, ISDN†, Analog†

DM3: T-1/E-1†, ISDN†, Analog†  
SS7†

†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The **gc\_GetParm( )** function retrieves the value of the specified parameter from the **parm\_id** parameter for a line device. The application can retrieve only one parameter value at a time.

See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>parm_id</b>	parameter ID definitions are listed in <a href="#">Table 16, “Parameter Descriptions, gc_GetParm( ) and gc_SetParm( )”</a> , on page 301 in the <a href="#">gc_SetParm( )</a> function description. The “Level” column lists whether the parameter is a channel level parameter or a trunk level parameter. To get a trunk level parameter, the <b>linedev</b> parameter must be the device ID associated with a network interface trunk.
<b>valuep</b>	points to the buffer where the requested information will be stored. See <a href="#">GC_PARM</a> , on page 391 for data structure details.

### ■ Termination Events

None

### ■ Cautions

None

## ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

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

int get_calling_party(LINEDEV ldev, char *callerp)
{
    GC_PARAM      parm_val;          /* value of parameter returned */
    GC_INFO       gc_error_info;     /* GlobalCall error information data */

    /*
     * 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_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetParm() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    return (0);
}
```

## ■ See Also

- [gc\\_SetParm\(\)](#)

## gc\_GetResourceH( )

**Name:** int gc\_GetResourceH(linedev, resourcehp, resourcetype)

**Inputs:**

LINEDEV linedev	• Global Call line device handle
int *resourcehp	• pointer to returned resource device handle
int resourcetype	• type of resource

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** voice and media, system controls and tools

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN, Analog

DM3: T-1/E-1, ISDN, Analog

IP (host-based stack† and embedded stack†)

†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The **gc\_GetResourceH()** function retrieves the resource device handle associated with the specified line device, **linedev**. The **\*resourcehp** parameter is actually the SRL handle of the resource associated with the line device.

If the **linedev** parameter is not an IP media Global Call line device, the resource to be retrieved can be a network, voice, or media device, as determined by the **resourcetype** parameter.

If the **linedev** parameter is an IP media Global Call line device, the resource that is retrieved is the Global Call line device to which the media handle is attached, as determined by a **resourcetype** parameter with a value of GC\_NET\_GCLINEDEVICE.

Parameter	Description
<b>linedev</b>	Global Call line device handle. This can be one of the following types: <ul style="list-style-type: none"> <li>• A non-IP media Global Call line device handle</li> <li>• An IP media Global Call line device handle</li> </ul>
<b>resourcehp</b>	address at which the associated resource device handle will be stored
<b>resourcetype</b>	type of resource. Possible values are: <ul style="list-style-type: none"> <li>• GC_NETWORKDEVICE</li> <li>• GC_VOICEDevice</li> <li>• GC_MEDIADevice</li> <li>• GC_NET_GCLINEDEVICE (only valid if the <b>linedev</b> parameter is an IP media Global Call line device handle)</li> </ul>



## ■ Termination Events

None

## ■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value `EGC_UNSUPPORTED` will be the Global Call value returned when the `gc_ErrorInfo()` function is used to retrieve the error code.

## ■ Errors

If this function returns `<0` to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

```
#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)
{
    GC_INFO      gc_error_info;    /* GlobalCall error information data */

    if (gc_GetResourceH(ldev, voicehp, GC_VOICEDevice) == 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_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetResourceH() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,\n",
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
}
```

## ■ See Also

- `gc_GetNetworkH()` (deprecated)
- `gc_GetVoiceH()` (deprecated)

## gc\_GetSigInfo( )

**Name:** int gc\_GetSigInfo(linedev, valuep, info\_id, metaeventp)

**Inputs:**

LINEDEV linedev	• Global Call line device handle
char *valuep	• pointer to info buffer
int info_id	• call info ID
METAEVENT *metaeventp	• pointer to metaevent block

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** interface specific

**Mode:** synchronous

**Platform and** Springware: ISDN†

**Technology:** DM3: ISDN†

SS7†

†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The **gc\_GetSigInfo( )** function retrieves the signaling information of an incoming message. The Global Call technology library uses **valuep** to retrieve the associated signaling information elements (IEs) and puts the information in the queue. To use the **gc\_GetSigInfo( )** function for a channel, the application needs to specify the size of the queue by calling the **gc\_SetParm( )** function and setting the RECEIVE\_INFO\_BUF to the desired size. See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>valuep</b>	points to buffer where call information is stored
<b>info_id</b>	identifies the type of signaling information to be retrieved; see Table 7
<b>metaeventp</b>	points to the METAEVENT structure filled in by the <b>gc_GetMetaEvent( )</b> function or the <b>gc_GetMetaEventEx( )</b> function (Windows extended asynchronous model only). See <b>METAEVENT</b> , on page 403 for more information.

**Table 7. `gc_GetSigInfo()` `info_id` Parameter ID Definitions**

<b>info_id Parameter</b>	<b>Definition</b>	<b>Technology</b>	<b>valuep Format</b>
U_IES	Unformatted user-to-user Information Elements (IEs) retrieved in CCITT format. The application needs to allocate sufficient memory (up to 256 bytes) to hold the retrieved IEs. The IEs are returned as raw data and must be parsed and interpreted by the application. Use the IE_BLK data structure to retrieve unprocessed IEs. For a description of the IE_BLK structure, and for DPNSS protocol-related IEs, see the <i>Global Call ISDN Technology User's Guide</i> .	ISDN, SS7	string
UUI	User-to-User Information. The data returned is application-dependent and is retrieved using the USRINFO_ELEM data structure. For a description of the return format for UUI and other technology-specific information, see the <i>Global Call ISDN Technology User's Guide</i> .	ISDN	string

### ■ Termination Events

None

### ■ Cautions

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.

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

### ■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <gcclib.h>
#include <gcerr.h>
#include <gcisdn.h>

#define MAX_QUEUE_SIZE    20

METAEVENT metaevent;
GC_PARM gcparm;

void main()
{
    LINEDEV ldev;
    char *devname = ":N_dtiB1T1:P_isdn";
    GC_INFO gc_error_info; /* GlobalCall error information data */
}
```

```

/*get line device handler */
if (gc_OpenEx(&ldev, devname, EV_SYNC, &ldev) != GC_SUCCESS)
{
    /* process error return as shown */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_OpenEx() on devname: %s, GC ErrorValue: 0x%hx - %s,
           CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
           devname, gc_error_info.gcValue, gc_error_info.gcMsg,
           gc_error_info.ccLibId, gc_error_info.ccLibName,
           gc_error_info.ccValue, gc_error_info.ccMsg);
    return (-1);
}

/* using gc_SetParm() to set size of the buffer queue for gc_GetSigInfo() */
gcparm.longvalue = MAX_QUEUE_SIZE;
if (gc_SetParm(ldev, RECEIVE_INFO_BUF, gcparm) != GC_SUCCESS)
{
    /* process error return as shown */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_SetParm() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
           CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
           ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
           gc_error_info.ccLibId, gc_error_info.ccLibName,
           gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}
}

/* Retrieve events from SRL */
int evt_hdlr()
{
    IE_BLK ie_blk;          /* Information Element */
    LINEDEV ldev;           /* Line device */
    CRN crn;                /* Call Reference Number */
    GC_INFO gc_error_info;   /* GlobalCall error information data */
    int retcode;             /* Return Code */

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

    crn = metaevent.crn;
    if (gc_CRN2LineDev(crn, &ldev) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_CRN2LineDev() on crn: 0x%lx, GC ErrorValue: 0x%hx - %s,
               CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
               crn, gc_error_info.gcValue, gc_error_info.gcMsg,
               gc_error_info.ccLibId, gc_error_info.ccLibName,
               gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    switch(metaevent.evtttype) {

case GCEV_CALLINFO:
    /* retrieve signaling information from queue */
    if (gc_GetSigInfo(ldev, (char *)&ie_blk, U_IES, &metaevent) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetSigInfo() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
               CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
               ldev, gc_error_info.gcValue, gc_error_info.gcMsg,

```

```

        gc_error_info.ccLibId, gc_error_info.ccLibName,
        gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}
else {
    /* succeeded, process signaling information */
}

break;
}
return 0;
}

```

#### ■ See Also

- [gc\\_GetCallInfo\(\)](#)
- [gc\\_SetParm\(\)](#)

## gc\_GetUserInfo( )

**Name:** int gc\_GetUserInfo (target\_type, target\_id, infoparmblkp)

**Inputs:** int target\_type                      • type of target object (line device or call reference number)  
           long target\_id                      • ID of target: either line device handle or call reference number

**Outputs:** GC\_PARM\_BLK\*                      • pointer to location of user information GC\_PARM\_BLK as  
           infoparmblkp                      defined by call control library

**Returns:** 0 if successful  
           <0 if failure

**Includes:** gclib.h

**Category:** interface specific, FTE

**Mode:** synchronous

**Platform and** Springware: ISDN†

**Technology:** †See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The **gc\_GetUserInfo( )** function retrieves technology-specific user information. An example of user information is information elements (IEs) in the case of ISDN. The actual definition and use of the user information is totally technology dependent. See the associated *Global Call Technology User's Guide* for definition and usage of the user information parameter.

For the **gc\_GetUserInfo( )** function, the target type identifies whether the basic entity is a line device (GCTGT\_GCLIB\_CHAN) or a call (GCTGT\_GCLIB\_CRN). See [Section 6.2, “Target Objects”](#), on page 417 for more information.

Parameter	Description
<b>target_type</b>	target object type. Valid values are: <ul style="list-style-type: none"> <li>• GCTGT_GCLIB_CHAN</li> <li>• GCTGT_GCLIB_CRN</li> </ul> See <a href="#">Table 22, “GC Parameter Entry List Maintained in GCLIB”</a> , on page 422 for details.
<b>target_id</b>	target object identifier. This identifier, along with <b>target_type</b> , uniquely specifies the target object. Valid identifiers are: <ul style="list-style-type: none"> <li>• line device handle</li> <li>• call reference number</li> </ul>
<b>infoparmblkp</b>	address of pointer to the location of <a href="#">GC_PARM_BLK</a> , which contains user information parameters. See the appropriate <i>Global Call Technology User's Guide</i> for definitions of the user information parameters contained in the GC_PARM_BLK.

## ■ Termination Events

None

## ■ Cautions

The user information parameters in the `GC_PARM_BLK` buffer that are returned via the **infoparmblkp** parameter must be processed or copied prior to the next Global Call function call. This is because the `GC_PARM_BLK` buffer will be deallocated in a subsequent Global Call function call.

## ■ Errors

If this function returns `<0` to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
#include <gcclib.h>
#include <gcerr.h>
#include <gcisdn.h>

/* ISDN: Retrieves and copies unprocessed information elements in CCITT format. */
/* Refer to "ISDN Technology User's Guide" for details. */
int GetUserInfo (LINEDEV linedev, IE_BLK **ie_blkp)
{
    GC_PARM_BLKP infoparmblkp = NULL; /* input parameter block pointer */
    int         retval = GC_SUCCESS;
    GC_INFO     gc_error_info; /* GlobalCall error information data */
    GC_PARM_DATAP t_parm_datap = NULL;

    /* retrieve the information element for the specified line device */
    retval = gc_GetUserInfo(GCTGT_GCLIB_NETIF, linedev, &infoparmblkp);

    if (retval != GC_SUCCESS)
    {
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetUserInfo() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                linedev, gc_error_info.gcValue, gc_error_info.gCMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    /* retrieve and copy the IE block from the parameter block */
    t_parm_datap = gc_util_find_parm(infoparmblkp, GCIS_SET_IE, GCIS_PARM_UIEDATA);
    memcpy((*ie_blkp), t_parm_datap, sizeof(IE_BLK));

    return (retval);
}
```

## ■ See Also

- `gc_SetUserInfo()`

## gc\_GetUsrAttr( )

**Name:** int gc\_GetUsrAttr(linedev, usr\_attrp)

**Inputs:** LINEDEV linedev      • Global Call line device handle  
void \*\*usr\_attrp      • pointer to location where user attribute info will be stored

**Returns:** 0 if successful  
<0 if failure

**Includes:** gcilib.h  
gcerr.h

**Category:** system controls and tools

**Mode:** synchronous

**Platform and Technology:** All

### ■ Description

The **gc\_GetUsrAttr( )** function retrieves the user-defined attribute established previously for the line device by the **gc\_SetUsrAttr( )** or **gc\_OpenEx( )** function.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>usr_attrp</b>	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 <b>gc_SetUsrAttr( )</b> or <b>gc_OpenEx( )</b> function.

### ■ Termination Events

None

### ■ Cautions

None

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).



## ■ Example

```
#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)
{
    GC_INFO gc_error_info; /* GlobalCall error information data */

    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_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetUsrAttr() on linedev: 0x%x, GC ErrorValue: 0x%x - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
                ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

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

## ■ See Also

- [gc\\_SetUsrAttr\(\)](#)
- [gc\\_OpenEx\(\)](#)

## gc\_GetVer()

**Name:** int gc\_GetVer(linedev, releasenum, inthump, component)

**Inputs:**

LINEDEV linedev	• Global Call line device handle
unsigned int *releasenum	• pointer to location where production release number will be stored
unsigned int *inthump	• 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 call handling

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN, Analog  
SS7  
IP (host-based stack and embedded stack)

---

### ■ Description

The **gc\_GetVer()** function retrieves the version number of a specified component. If the specified component parameter is GCGV\_LIB or if the **linedev** parameter is 0 (zero), the version number of the Global Call library will be returned. Otherwise, the value returned will be the version number of the library associated with the **linedev** parameter.

An Intel® Dialogic version number consists of two parts:

- the release type: production or beta
- the release number, which consists of different elements depending on the type of release

For example:

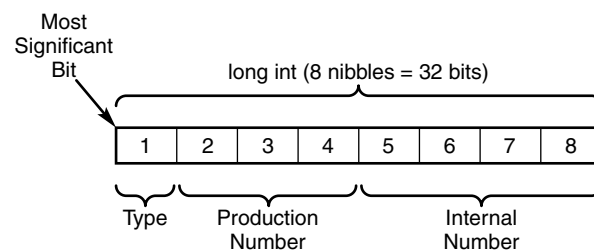
- 1.00 Production
- 1.00 Beta 5

Parameter	Description
<b>linedev</b>	Global Call line device handle. If this parameter is set to 0, the version number of the Global Call API is returned.
<b>releasenum</b>	points to the location where the production release number and type indicator will be stored

Parameter	Description
<b>intnump</b>	points to the location where the internal release number will be stored
<b>component</b>	specifies the software component to which the version number applies. Selections are: <ul style="list-style-type: none"> <li>• GC_LIB – Global Call library</li> <li>• GC_ICAPI_LIB – ICAPI library</li> <li>• GC_ANAPI_LIB – ANAPI library</li> <li>• GC_PDKRT_LIB – PDKRT library</li> </ul>

The `gc_GetVer()` function returns the software version number as a long integer (32 bits) in BCD (binary coded decimal) format. Figure 1 shows the format of the version number that is returned. Each section in the diagram represents a nibble (4 bits).

**Figure 1. Component Version Number Format**



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

- 0 - Production
- 1 - Beta
- 2 - Alpha
- 3 - Experimental

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

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.

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

Table 8 provides the values returned by each nibble in the long int. For example, if a production version number is 1.02, then:

- `*releasenum = 0x0102`
- `*intnum = 0x0000`

For a version number of 1.02 beta 2, then:

- `*releasenum = 0x1102`

- \*intnum = 0x0200

**Table 8. gc\_GetVer( ) Return Values**

*releasenum			*intnum	
1†	2†	3 & 4†	5 & 6†	7 & 8†
Type	Production Number		Internal Number	
Production	Major Production Number	Minor Production Number	N/A	N/A
Beta	Major Production Number	Minor Production Number	Beta Number	N/A
† Nibble(s) [4 bits each]				

## ■ Termination Events

None

## ■ Cautions

None

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
#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 */
    GC_INFO         gc_error_info;     /* GlobalCall error information data */

    /*
     * 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_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetVer() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",

```

```

        ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
        gc_error_info.ccLibId, gc_error_info.ccLibName,
        gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}

return (0);
}

```

#### ■ See Also

None

## gc\_GetVoiceH( )

**Name:** int gc\_GetVoiceH(linedev, voicehp)

**Inputs:** LINEDEV linedev      • Global Call line device handle  
int \*voicehp      • pointer to returned voice device handle

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** voice and media

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN, Analog

DM3: T-1/E-1, ISDN, Analog  
SS7  
IP (embedded stack)

### ■ Description

**Note:** For technologies that support the [gc\\_GetResourceH\( \)](#) function, it is recommended that the [gc\\_GetResourceH\( \)](#) function be used instead of the [gc\\_GetVoiceH\( \)](#) function to retrieve the voice device handle. See the [gc\\_GetResourceH\( \)](#) function description for more information.

The [gc\\_GetVoiceH\( \)](#) function retrieves 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.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>voicehp</b>	address at which the device handle of the voice resource associated with the Global Call line device, <b>linedev</b> , will be stored

### ■ Termination Events

None

### ■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value EGC\_UNSUPPORTED will be the Global Call value returned when the [gc\\_ErrorInfo\( \)](#) function is used to retrieve the error code.

## ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

```
#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)
{
    GC_INFO      gc_error_info;    /* GlobalCall error information data */

    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_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetVoiceH() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,\n",
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
}
```

## ■ See Also

- [gc\\_GetResourceH\(\)](#)
- [gc\\_GetNetworkH\(\)](#) (deprecated)

## gc\_GetXmitSlot( )

**Name:** int gc\_GetXmitSlot(linedev, sctsinfop)

**Inputs:** LINEDEV linedev      • Global Call line device handle  
SC\_TSINFO \*sctsinfop      • pointer to the CT Bus time slot information structure

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** system controls and tools

**Mode:** synchronous

**Platform and** Springware: T-1/E-1 (PDKRT only)

**Technology:** DM3: T-1/E-1, ISDN, Analog  
SS7

IP (host-based stack† and embedded stack†)

†See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

The **gc\_GetXmitSlot()** function retrieves the network CT Bus time slot number of the specified device's channel. The CT Bus time slot information is returned in the [SC\\_TSINFO](#) structure, which includes the number of the network CT Bus time slot connected to the device's transmit channel.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>sctsinfop</b>	points to the SC_TSINFO CT Bus time slot information structure. See <a href="#">SC_TSINFO</a> , on page 405 for more information.

### ■ Termination Events

None

### ■ Cautions

For voice resources, the routing function for the corresponding library must be called, for example, **dx\_getxmitslot()**, **dx\_listen()**, and **dx\_unlisten()**.

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology



specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

### ■ Example

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

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

/*
 * Data structure which stores all information for each line
 */
static struct linebag {
    LINEDEV      ldev;      /* GlobalCall API line device handle */
    CRN          crn;       /* GlobalCall API call handle */
    int          blocked;   /* channel blocked/unblocked */
    int          networkh;  /* network handle */
} port[MAX_CHAN+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 and voice handle is stored in
 *       linebag structure "port"
 */
int call_getxmitslot(int port_num)
{
    GC_INFO      gc_error_info;  /* GlobalCall error information data */
    SC_TSINFO    sc_tsinfo;      /* CTBus timeslot structure */
    long         scts;

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

    /* Fill in the CTbus time slot information */
    sc_tsinfo.sc_numts = 1;
    sc_tsinfo.sc_tsarrayp = &scts;

    /* Get CTbus time slot connected to transmit of
     * digital channel on board ...1 */
    if (gc_GetXmitSlot(pline->ldev, &sc_tsinfo) == -1)
    {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetXmitSlot() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                pline->ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    return (0);
}
```

### ■ See Also

- [gc\\_GetCTInfo\(\)](#)

***gc\_GetXmitSlot()*** — retrieve the network CT Bus time slot number



- [gc\\_Listen\(\)](#)
- [gc\\_UnListen\(\)](#)

## gc\_HoldACK( )

**Name:** int gc\_HoldACK(crn)

**Inputs:** CRN crn • call reference number

**Returns:** 0 if successful  
<0 if failure

**Includes:** gcLib.h  
gcerr.h

**Category:** interface specific

**Mode:** synchronous

**Platform and Technology:** Springware: ISDN BRI, ISDN PRI (DPNSS and Q.SIG only)

---

### ■ Description

The **gc\_HoldACK( )** function accepts a hold request from remote equipment. Call this function only when the call is in the Connected state and after the GCEV\_HOLDCALL event is received.

Parameter	Description
crn	call reference number

### ■ Termination Events

None

### ■ Cautions

The call must be in the Connected state and the GCEV\_HOLDCALL event detected before this function is invoked or the function will fail.

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

**■ 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) has been
 *    called to convert the event into metaevent.
 * 4. a GCEV_OFFERED event has been detected.
 */

#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>
METAEVENT metaevent;

int hold_call()
{
    LINEDEV  ldev;          /* Line device */
    CRN      crn;           /* call reference number */
    GC_INFO  gc_error_info; /* GlobalCall error information data */

    /*
     * Do the following:
     * 1. Get the CRN from the metaevent
     * 2. Proceed to place the call on hold
     */

    crn = metaevent.crn;

    if(gc_HoldCall(crn, EV_ASYNC) != GC_SUCCESS) {
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_HoldCall() on linedev: 0x%x, GC ErrorValue: 0x%x - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
                metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    return 0;
}

/* Retrieve events from SRL */
int evt_hdlr()
{
    LINEDEV  ldev;          /* Line device */
    CRN      crn;           /* Call Reference Number */
    GC_INFO  gc_error_info; /* GlobalCall error information data */
    int      retcode;       /* Return Code */

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

    crn = metaevent.crn;
}
```

```
switch(metaevent.evtttype) {
case GCEV_HOLDCALL:
    /* retrieve signaling information from queue */
    if (gc_HoldACK(crn) != GC_SUCCESS) {
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_HoldACK() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
    }
    else {
        /* succeeded, process signaling information */
    }
    break;
}
return 0;
}
```

#### ■ See Also

- [gc\\_HoldCall\(\)](#)
- [gc\\_RetrieveCall\(\)](#)

## **gc\_HoldCall( )**

**Name:** int gc\_HoldCall(crn, mode)

**Inputs:** CRN crn • call reference number for the active call  
unsigned long mode • async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** advanced call model

**Mode:** asynchronous or synchronous

**Platform and Technology:** Springware: T-1/E-1 (PDKRT only)†, ISDN BRI, ISDN PRI (DPNSS and Q.SIG only)†  
SS7†  
†See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

The **gc\_HoldCall( )** function places an active call on hold. The call must be in the Connected state to be put on hold.

Parameter	Description
<b>crn</b>	call reference number
<b>mode</b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

### ■ Termination Events

#### **ISDN events:**

##### **GCEV\_HOLDACK**

indicates that the **gc\_HoldCall( )** function was successful, that is, the call was placed on hold.

##### **GCEV\_HOLDREJ**

indicates that the function failed and the request to place the call on hold was rejected.

#### **PDKRT events:**

##### **GCEV\_HOLDCALL**

indicates that the **gc\_HoldCall( )** function was successful, that is, the call was placed on hold.

##### **GCEV\_TASKFAIL**

indicates that the function failed. See the “Error Handling” section in the *Global Call API Programming Guide*.

## ■ Cautions

The **gc\_HoldCall()** function should only be used when the call is in the Connected state or the function will fail.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function for error information. If the GCEV\_HOLDREJ or GCEV\_TASKFAIL event is received, use the **gc\_ResultInfo()** function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ 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) has been
 *    called to convert the event into metaevent.
 * 4. a GCEV_OFFERED event has been detected.
 */
#include <stdio.h>
#include <srllib.h>
#include <gcLib.h>
#include <gcerr.h>
#include <gcisdn.h>
METAEVENT metaevent;

int hold_call()
{
    LINEDEV    ldev;          /* Line device */
    CRN        crn;          /* call reference number */
    GC_INFO    gc_error_info; /* GlobalCall error information data */

    /*
     * Do the following:
     * 1. Get the CRN from the metaevent
     * 2. Proceed to place the call on hold
     */

    crn = metaevent.crn;

    if (gc_HoldCall(crn, EV_ASYNC) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_HoldCall() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    return 0;
}
```

■ **See Also**

- [gc\\_HoldACK\( \)](#)
- [gc\\_HoldRej\( \)](#)
- [gc\\_RetrieveCall\( \)](#)



## gc\_HoldRej()

**Name:** int gc\_HoldRej(crn, cause)

**Inputs:** CRN crn                      • call reference number  
          int cause                    • standard ISDN Network error code

**Returns:** 0 if successful  
          <0 if failure

**Includes:** gclib.h  
          gcerr.h

**Category:** interface specific

**Mode:** synchronous

**Platform and Technology:** Springware: ISDN BRI, ISDN PRI (DPNSS and Q.SIG only)

---

### ■ Description

The **gc\_HoldRej()** function rejects a hold request from remote equipment. This function is called only after the call is in the Connected state and after the GCEV\_HOLDCALL event is received.

Parameter	Description
<b>crn</b>	call reference number
<b>cause</b>	a standard ISDN Network cause/error code is returned indicating the reason the hold request was rejected. Possible causes include TEMPORARY_FAILURE (Cause 41), NETWORK_OUT_OF_ORDER (Cause 38), and NETWORK_CONGESTION (Cause 42). For a complete listing of ISDN Network cause/error codes, see the <i>Global Call ISDN Technology User's Guide</i> .

### ■ Termination Events

None

### ■ Cautions

- Call the **gc\_HoldRej()** function only after the call is in the Connected state and after receiving the GCEV\_HOLDCALL event or the function will fail.
- Not all ISDN Network cause/error codes are universally supported across switch types. Before you use a particular cause code, compare its validity with the appropriate switch vendor specifications.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ 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) has been
 *    called to convert the event into metaevent.
 * 4. a GCEV_OFFERED event has been detected.
 */
GC_INFO gc_error_info; /* GlobalCall error information data */

/*
 * Do the following:
 * 1. Get the CRN from the metaevent
 * 2. Proceed to place the call on hold
 */

crn = metaevent.crn;

if(gc_HoldCall(crn, EV_ASYNC) != GC_SUCCESS) {
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_HoldCall() on linedev: 0x%x, GC ErrorValue: 0x%x - %s,
            CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
            metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
            gc_error_info.ccLibId, gc_error_info.ccLibName,
            gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}

return 0;
}

/* Retrieve events from SRL */
int evt_hdlr()
{
    LINEDEV ldev; /* Line device */
    CRN crn; /* Call Reference Number */
    GC_INFO gc_error_info; /* GlobalCall error information data */
    int retcode; /* Return Code */

    retcode = gc_GetMetaEvent(&metaevent);
    if (retcode != GC_SUCCESS ) {
        /* get and process the error */
    } else {
        /* Continue with normal processing */
    }

    crn = metaevent.crn;

    switch(metaevent.evttype) {
    case GCEV_HOLD_CALL:
        /* retrieve signaling information from queue */
        if ( gc_HoldRej(crn, TEMPORARY_FAILURE) != GC_SUCCESS)
        {

```

```

        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ( "Error: gc_HoldRej() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
                  CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                  metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
                  gc_error_info.ccLibId, gc_error_info.ccLibName,
                  gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    else {
        /* succeeded, process signaling information */
    }

    break;
}
return 0;
}

```

#### ■ See Also

- [gc\\_HoldCall\(\)](#)
- [gc\\_RetrieveCall\(\)](#)

## **gc\_LinedevToCCLIBID( )**

**Name:** int gc\_LinedevToCCLIBID(linedev, cclibid)

**Inputs:** LINEDEV linedev • line device  
int \*cclibid • pointer to location of library identification code

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** library information

**Mode:** synchronous

**Platform and** All

**Technology:**

### ■ Description

The **gc\_LinedevToCCLIBID( )** function returns the call control library ID of the call control library that opened the specified line device.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>cclibid</b>	points to destination for call control library ID associated with this line device

### ■ Termination Events

None

### ■ Cautions

None

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

### ■ Example

```
#include <stdlib.h>
#include <stdio.h>
#include <gcLib.h>
#include <gcerr.h>

/*
 * This procedure prints the cclib information associated with the specified linedev
 */
void print_cclibid_info(LINEDEV linedev)
{
    int          cclibid;

    if (gc_LinedevToCCLIBID(linedev, &cclibid) == GC_SUCCESS) {
        printf("The cclibid for linedev %ld is %d\n", linedev, cclibid);
    } else {
        /* Do error handling */
    }
}
```

### ■ See Also

- [gc\\_CCLibIDToName\(\)](#)
- [gc\\_OpenEx\(\)](#)

## gc\_Listen()

**Name:** int gc\_Listen(linedev, sctsinfop, mode)

**Inputs:**

LINEDEV linedev	• Global Call line device handle
SC_TSINFO *sctsinfop	• pointer to CT Bus time slot information structure
unsigned long mode	• async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** system controls and tools

**Mode:** asynchronous or synchronous

**Platform and Technology:** Springware: T-1/E-1 (PDKRT only)

DM3: T-1/E-1, ISDN, Analog  
SS7  
IP (host-based stack† and embedded stack†)  
†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The **gc\_Listen()** function connects a channel to a network CT Bus time slot. The **gc\_Listen()** function uses the information stored in the **SC\_TSINFO** structure to connect the receive channel on the device to an available network CT Bus time slot. The time slot number is returned in the **SC\_TSINFO** structure. When the **gc\_Listen()** function returns, the receive channel is connected to the CT Bus time slot.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>sctsinfop</b>	points to the <b>SC_TSINFO</b> CT Bus time slot information structure. See <a href="#">SC_TSINFO</a> , on page 405 for more information.
<b>mode</b>	set to <b>EV_ASYNC</b> for asynchronous execution or to <b>EV_SYNC</b> for synchronous execution

### ■ Termination Events

**GCEV\_LISTEN**  
indicates that a time slot has been routed successfully.

**GCEV\_TASKFAIL**  
indicates that the time slot routing has failed.

## ■ Cautions

For voice resources, the routing function for the corresponding library must be called, for example, **dx\_getxmitslot()**, **dx\_listen()**, and **dx\_unlisten()**.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

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

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

/*
 * Data structure which stores all information for each line
 */
static struct linebag {
    LINEDEV  ldev;          /* GlobalCall API line device handle */
    CRN      crn;           /* GlobalCall API call handle */
    int      blocked;       /* channel blocked/unblocked */
    int      networkh;      /* network handle */
    int      chdev;         /* voice handle */
} port[MAX_CHAN+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. Opened a voice device on dxxxB1C1.
 *   3. Each line device and voice handle is stored in linebag structure "port"
 */
int call_listen(int port_num)
{
    GC_INFO      gc_error_info; /* GlobalCall error information data */
    SC_TSINFO    sc_tsinfo;     /* CTbus time slot structure */
    long scts;

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

    /* Fill in the CTbus time slot information */
    sc_tsinfo.sc_numts = 1;
    sc_tsinfo.sc_tsarrayp = &scts;

    /* Get CTbus time slot connected to transmit of voice
       channel on board ...1 */
    if (dx_getxmitslot(pline->ldev, &sc_tsinfo) == -1)
    {
        printf("Error message = %s", ATDV_ERRMSGP(pline->ldev));
        exit(1);
    }
}
```

```
/* Connect the receive of the digital channel 1 on board 1 to CTBus
time slot of voice channel 1 */
if (gc_Listen(pline->ldev, &sc_tsinfo, EV_SYNC) == -1)
{
    /* process error return as shown */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_Listen() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
           CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
           pline->ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
           gc_error_info.ccLibId, gc_error_info.ccLibName,
           gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}

return (0);
}
```

■ **See Also**

- [gc\\_GetCTInfo\(\)](#)
- [gc\\_GetXmitSlot\(\)](#)
- [gc\\_UnListen\(\)](#)



## gc\_LoadDxParm( )

**Name:** int gc\_LoadDxParm(linedev, pathp, errmsgp, err\_length)

**Inputs:** LINEDEV linedev                      • line device  
char \*pathp                                      • pointer to parameter file  
char \*errmsgp                                   • pointer to error message  
int err\_length                                   • maximum error message length

**Returns:** 0 if successful  
             <0 if failure

**Includes:** gclib.h  
             gcerr.h

**Category:** voice and media

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1 (PDKRT only), Analog

### ■ Description

The **gc\_LoadDxParm()** function sets voice parameters associated with a line device that operates as a dedicated or shared resource. The line device resource operates 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.

Global Call assigns an 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\_OpenEx()** function is called. This identification number assignment remains valid until the **gc\_Close()** function is called to close the line devices.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>pathp</b>	points to the voice parameter file to load
<b>errmsgp</b>	points to the storage address of any error message
<b>err_length</b>	maximum length in bytes of the error message stored at address defined by the <b>errmsgp</b> parameter

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 **pathp** parameter, typically in the current directory or in the standard 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 Voice Software Reference for the DX_CAP
#   structure ca_* parameters. The upper case parameters can also
#   be found in the Voice Software Reference - Features Guide 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
```

A voice parameter file contains parameter definition lines and may contain comment lines. Each parameter definition line comprises a case-sensitive voice parameter as the first field of the line, a space, and a second field defining the parameter value.

A comment line begins with either of the following characters:

- # character
- ; character

The `gc_LoadDxParm()` function will return upon the first detected error. The reason for the error will be stored in the `msgbufferp` location. Typical error reasons are:

- a parsing error (in the `.vcp` file)
- a low-level function call error
- an open file failure error

**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 set up by the `gc_MakeCall()` function.

All channel-level parameters set by the voice function, `dx_setparm()`, can be set using the `gc_LoadDxParm()` function. Global Call uses the `dx_setparm()` parameter names to identify all voice channel-level parameters:

- `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`

Also see the *Voice API Library Reference* for your operating system 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 parameter value may be entered as a decimal value or as a hexadecimal value when prefixed with a "0x". The call analysis parameters defined in the

DX\_CAP data structure affect the [gc\\_MakeCall\(\)](#) function. Global Call uses the DX\_CAP data structure names to identify all call progress parameters:

- ca\_allowmax
- 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\_higlth
- 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\_loglth
- ca\_lower2frq
- ca\_lower3frq
- ca\_lowerfrq
- ca\_maxansr
- ca\_maxintering
- ca\_mxtime2frq
- ca\_mxtime3frq
- ca\_mxtimefrq
- ca\_nrbeg
- 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`

Also see the *Voice API Library Reference* for your operating system for more information about the fields in the [DX\\_CAP](#) data structure.

For analog applications, the `gc_LoadDxParm()` function is used to set call analysis parameters (board and channel-level) that are otherwise 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.

For technologies other than analog, a voice resource must be attached before calling this function.

## ■ Termination Events

None

## ■ Cautions

- Before calling the `gc_LoadDxParm()` function, the `gc_OpenEx()` function must be used to open the voice line device.
- The maximum length of the error message string, **msglength**, must be passed to the `gc_LoadDxParm()` function to avoid overwriting memory locations outside the message string array.
- 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 above) are used only 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 Global Call value returned when the `gc_ErrorInfo()` function is used to retrieve the error code. See also the [Errors](#) section of this function description.

- When using the North America Analog Bidirectional Protocol for ANAPI, the application **must** call the **gc\_LoadDxParm()** function after calling **gc\_OpenEx()**. Failure to do so may lead to incorrect call progress information.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

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.

## ■ Example

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

#define MSGLENGTH 80

main()
{
    LINEDEV ldev;

    GC_INFO gc_error_info; /* GlobalCall error information data */
    .
    .
    .
    /*
     * Assume the following has been done:
     * Open line device (ldev) specifying voice and network resource using
     * gc_OpenEx()
     */
    /* call gc_LoadDxParm() to download the channel parameters */
    if ((gc_LoadDxParm(ldev, "dxchan.vcp", errmsg, MSGLENGTH)) != 0) {
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_LoadDxParm() on linedev: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    .
    .
    .
}
```

## ■ See Also

- **gc\_MakeCall()**



*set voice parameters associated with a line device — `gc_LoadDxParm()`*

- `gc_OpenEx()`

**gc\_MakeCall( )****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**Category:** basic**Mode:** asynchronous or synchronous**Platform and** All†**Technology:** †See the *Global Call Technology User's Guides* for additional information.**■ Description**

The **gc\_MakeCall( )** function makes an outgoing call on the specified line device. When this function is issued asynchronously, a call reference number (CRN) will be assigned and returned immediately if the function is successful. All subsequent communications between the application and the Global Call 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.

The **gc\_MakeCall( )** function changes the call state from the Null state or, when performing a supervised transfer, the Dialtone state to the Dialing state. If the call is answered, the call state changes from the Dialing state to the Connected state. If the other end is not ready to receive the call, the call state changes from Dialing to Alerting. If the **gc\_MakeCall( )** function fails, the resulting call state depends on the point in the calling process where the failure occurred and the call control library being used.

See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>crnp</b>	points to the memory location where the call reference number is to be stored
<b>numberstr</b>	called party's telephone number (must be terminated with "\0"). Maximum length is 32 digits.



Parameter	Description
<b>makecallp</b>	points to the GC_MAKECALL_BLK structure; see <a href="#">GC_MAKECALL_BLK</a> , on page 390 for details. The GC_MAKECALL_BLK structure lists the parameters used to make an outbound call. Assigning a NULL to the <b>makecallp</b> parameter indicates that the default values should be used for the call.
<b>timeout</b>	time interval (in seconds) during which the call must be established, or the 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. (For DM3 boards, <b>timeout</b> is supported in both the synchronous and asynchronous modes.)  See the appropriate <i>Global Call Technology User's Guide</i> for technology-specific information.
<b>mode</b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

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\\_ReleaseCallEx\(\)](#) 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 must issue [gc\\_DropCall\(\)](#) and [gc\\_ReleaseCallEx\(\)](#) functions to terminate the call completely before issuing another [gc\\_MakeCall\(\)](#) function.

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 time-out 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 (for example, the ISDN library).

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

Table 9 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\\_ReleaseCallEx\(\)](#) functions before you can initiate your next call.
- Synchronous: When an error condition is encountered, a value that is <0 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\\_ReleaseCallEx\(\)](#) 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.

**Table 9. Call Conditions and Results**

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

## ■ Termination Events

### GCEV\_CALLSTATUS

indicates that a time-out or no answer condition was returned while the [gc\\_MakeCall\(\)](#) function was active.

### GCEV\_CONNECTED

indicates that the [gc\\_MakeCall\(\)](#) function was successful, that is, the call resulted in a connection.

**GCEV\_DISCONNECTED**

indicates that a request or message was rejected or that it timed out, halting further processing of the call.

**GCEV\_TASKFAIL**

indicates that the function failed. For more information, see the “Error Handling” section in the *Global Call API Programming Guide*.

**Note:** See Table 9 for other events that may be received by the application.

## ■ Cautions

- In both asynchronous and synchronous modes, after a time-out 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.
- If the **timeout** parameter is set to a value larger than a protocol time-out value, a protocol time-out may occur first, which will cause the `gc_MakeCall()` function to fail. For ANAPI and ICAPI protocols, the protocol time-out is configured in the CDP file.

## ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

```
#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)
{
    GC_INFO      gc_error_info;  /* GlobalCall error information data */

    /* 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_ErrorInfo( &gc_error_info );
    printf ("Error: gc_MakeCall() on linedev: 0x%x, GC ErrorValue: 0x%x - %s,
           CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
           pline->ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
           gc_error_info.ccLibId, gc_error_info.ccLibName,
           gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}

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

#### ■ See Also

- [gc\\_DropCall\(\)](#)
- [gc\\_GetCallInfo\(\)](#)
- [gc\\_LoadDxParm\(\)](#)
- [gc\\_ReleaseCallEx\(\)](#)

## gc\_Open( )

**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 (deprecated)

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN, Analog  
DM3: T-1/E-1, ISDN, Analog  
SS7  
IP (embedded stack)

---

### ■ Description

**Note:** The **gc\_Open()** function is deprecated in this software release. The **gc\_OpenEx()** function replaces this function and must be used instead.

The **gc\_Open()** function opens a Global Call device and returns a unique line device ID (or handle) to identify the physical device or devices that carry the call.

## gc\_OpenEx( )

**Name:** int gc\_OpenEx(linedevp, devicename, mode, usrattr)

**Inputs:**

LINEDEV *linedevp	• pointer to returned line device
char *devicename	• pointer to ASCII string
int mode	• async or sync
void *usrattr	• pointer to user attribute

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** system controls and tools

**Mode:** asynchronous or synchronous

**Platform and** All†

**Technology:** †See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The **gc\_OpenEx( )** function opens a Global Call device and sets a user-defined attribute. The function also returns a unique line device ID (or handle) to identify the physical device or devices that carry the call. For example, a line device may represent a single network or time slot, or the grouping together of a time slot and a voice or media channel. All subsequent references to the opened device must be made using the line device ID. Both network board and channel (time slot) devices can be opened using the **gc\_OpenEx( )** function. A device may be opened only once and cannot be re-opened by the current process or by any other process until the device is closed.

After the successful return of the **gc\_OpenEx( )** function, the application must wait for a GCEV\_UNBLOCKED event before proceeding with a call (make call or wait call) on the opened line device. When the GCEV\_UNBLOCKED event is received, then the line is ready to accept or make calls. Note that the GCEV\_UNBLOCKED event may be received before **gc\_OpenEx( )** returns; the application must be prepared to handle this.

The **gc\_OpenEx( )** function should 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 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 or line device identification.

Parameter	Description
<b>linedevp</b>	points to unique number to be filled in by this function to identify a specific device
<b>devicename</b>	<p>points to an ASCII string that defines the device(s) associated with the returned <b>linedevp</b> number. The <b>devicename</b> parameter specifies the device to be opened and the protocol to be used.</p> <p>The format used to define <b>devicename</b> is:</p> <p>&lt;field1&gt;&lt;field2&gt;...&lt;fieldn&gt;</p> <p>These fields may be listed in any order. The field format is:</p> <p>:&lt;key&gt;_&lt;field name&gt;</p> <p>Valid keys and their appropriate field names are:</p> <ul style="list-style-type: none"> <li>• <b>P</b> - <i>protocol_name</i> – specifies the protocol to be used. See the appropriate <i>Global Call Technology User's Guide</i> for technology specific protocol information.</li> </ul> <p>The <i>protocol_name</i> field is not used on DM3 boards using T-1 or ISDN technology, because the protocol is determined at board initialization time and not when a Global Call device is opened.</p> <ul style="list-style-type: none"> <li>• <b>N</b> - <i>network_device_name</i> – specifies the board name and the time slot name (if needed) using the following naming convention:</li> </ul> <p>If the board is to be opened, the <i>network_device_name</i> is the board name. An example of the format is: <b>dtiB&lt;number of board&gt;</b></p> <p>If a time slot is to be opened, both the board and time slot are specified. An example of the format is: <b>dtiB&lt;number of board&gt;T&lt;time slot number&gt;</b></p> <p>See the appropriate <i>Global Call Technology User's Guide</i> for the correct <i>network_device_name</i>.</p> <ul style="list-style-type: none"> <li>• <b>V</b> - <i>voice_device_name</i> – specifies the voice board and channel. An example of the format is: <b>dxxxB&lt;virtual board number&gt;C&lt;channel number&gt;</b></li> </ul> <p>Attachment to different types of DM3 voice devices is dependent on the protocol downloaded. For example, if one board has ISDN for protocols and another has T-1 CAS, the T-1 CAS network devices cannot be attached to the voice devices on the ISDN board. See the <i>Global Call API Programming Guide</i> for further information.</p> <ul style="list-style-type: none"> <li>• <b>M</b> - <i>media_device_name</i> – specifies the media board and channel. An example of the format is: <b>ipmB&lt;virtual board number&gt;C&lt;channel number&gt;</b></li> </ul> <p>Not all technologies support voice or media devices. See the appropriate <i>Global Call Technology User's Guide</i> for the correct voice or media device names, if supported.</p>

Parameter	Description
<b>mode</b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution
<b>usrattr</b>	points to buffer where a user defined attribute is stored

## ■ Termination Events

### GCEV\_OPENEX

indicates successful completion of the **gc\_OpenEx()** function, that is, the device was opened.

### GCEV\_OPENEX\_FAIL

indicates that the **gc\_OpenEx()** function failed and the device was not opened. A **gc\_Close()** function must still be performed on the line device handle to free resources.

## ■ Cautions

- If a handler is enabled for the GCEV\_UNBLOCKED event, then the **linedevp** parameter passed to the **gc\_OpenEx()** function must be global so that any and all events can be processed.
- If a blocking alarm condition exists when opening a device, the application will not receive a GCEV\_BLOCKED event since the application is already blocked.
- When using **gc\_OpenEx()** in asynchronous mode to open a line device, an application must wait for the GCEV\_OPENEX termination event before any other Global Call functions can be used on that line device.
- To handle error returns from the **gc\_OpenEx()** function, use the Global Call error handling functions, **gc\_ErrorInfo()** and **gc\_ResultInfo()**. Do not use the Linux errno variable to get Global Call error information. If you get either an EGC\_DXOPEN or an EGC\_DTOPEN error, use the error handling procedures recommended in the *Standard Runtime Library API Programming Guide*, *Standard Runtime Library API Library Reference*, and *Digital Network Interface Software Reference* for your operating system.
- See the *Global Call Technology User's Guide* for the network interface being used to determine required **devicename** components and features unique to the network interface.
- Once a voice, media, or network device is opened, you must perform a **gc\_Close()** before calling **gc\_OpenEx()** again. Calling **gc\_OpenEx()** twice on the same device without first closing the device will not return an error, but any subsequent behavior of the resulting line device(s) is undefined.
- When using **gc\_OpenEx()** in asynchronous mode, if a line device cannot be opened and the GCEV\_OPENEX\_FAIL event is received, the application must use **gc\_Close()** on the line device handle to free resources.
- When using the North America Analog Protocol, the application **must** call the **gc\_LoadDxParm()** function after calling **gc\_OpenEx()**. Failure to do so may lead to incorrect call progress information.
- A GCEV\_UNBLOCKED event will be generated when opening a board device. A GCEV\_BLOCKED event will also be generated if there are blocking alarms on the board, and the corresponding GCEV\_UNBLOCKED event will be generated when the blocking alarms clear. The application must be prepared to handle these events.
- When using DM3 boards in a fixed routing configuration (see the “Working with Fixed Routing Configurations” section in the *Global Call API Programming Guide*), opening a



network device also results in opening a voice device that is permanently bound to the network device. An application should not attempt to open the voice device that is already attached to an opened network device. If an application opens a voice device that is already attached to a network device, the application will receive a duplicate of each event on the voice device.

## ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. If the `GCEV_OPENEX_FAIL` event is received, use the `gc_ResultInfo()` function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

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

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

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

/*
 * Global variable
 */
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 linebag {
    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;
```

```

/* 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_enbhdtr(EV_ANYDEV, EV_ANYEVT,
    (long *) (void *) evt_hdlr) == -1) {
    printf("sr_enbhdtr 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_waitvt(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)
{
    GC_INFO      gc_error_info; /* GlobalCall error information data */

    gc_ErrorInfo( &gc_error_info );
    printf( "Error: %s(), GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s,
        CC ErrorValue: 0x%lx - %s\n",
        function, gc_error_info.gcValue, gc_error_info.gcMsg,
        gc_error_info.ccLibId, gc_error_info.ccLibName,
        gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}

/*
 * int evt_hdlr (void)
 *
 * RETURN: 0          - function successful
 *         error      - GlobalCall error number
 *
 */
int evt_hdlr(void)
{
    struct channel    *pline;
    int               error;
    METAEVENT         metaevent;

    /* 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.evtttype) {
        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_OpenEx() function */
    int         vnum = 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 OpenEx 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++) {

        vnum = (ts - 1) / 4 + 1;
        vch = ((ts - 1) % 4) + 1;
        sprintf (devname, "N_dtiB1T%d:P_ar_r2_o:V_dxxxB%dC%d", ts, vnum, vch);
        sr_hold();
        if (gc_OpenEx(&port[port_index].ldev, devname, EV_SYNC,
                     (void *)&port[port_index]) != GC_SUCCESS) {
            /* process error return as shown */
            error = print_error("gc_OpenEx");
            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_GetResourceH(port[port_index].ldev,
    &(port[port_index].networkh, GC_NETWORKDEVICE)) != GC_SUCCESS) {
    /* process error return as shown */
    error = print_error("gc_GetResourceH(GC_NETWORKDEVICE)");
    sr_release();
    return(error);
}

if (gc_GetResourceH(port[port_index].ldev,
    &(port[port_index].voiceh), GC_VOICEDEVICE) != 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 0;
}

```

#### ■ See Also

- [gc\\_GetUsrAttr\(\)](#)
- [gc\\_SetUsrAttr\(\)](#)

## `gc_QueryConfigData()`

**Name:** `int gc_QueryConfigData (target_type, target_id, source_datap, query_id, response_datap)`

**Inputs:**

<code>int target_type</code>	• target object type
<code>long target_id</code>	• target object ID
<code>GC_PARM *source_datap</code>	• pointer to source data
<code>long query_id</code>	• query ID
<code>GC_PARM *response_datap</code>	• pointer to the result of the query

**Returns:** 0 if successful  
<0 if failure.

**Includes:** `gcerr.h`

**Category:** RTCM

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1 (PDKRT only)†, ISDN

†See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

The `gc_QueryConfigData()` function supports the Global Call Run Time Configuration Management (RTCM) feature. Given the **query\_id**, the `gc_QueryConfigData()` function queries the configuration data based on the source data from a target object. See [Section 6.2, “Target Objects”](#), on page 417 for more information.

The `gc_QueryConfigData()` function can be used to obtain parameter information via its name (for example, protocol ID from protocol name or parameter ID from parameter name). Once the parameter information is obtained, the `gc_SetConfigData()` and `gc_GetConfigData()` functions can be called to update and retrieve the configuration data, respectively.

For more information about the RTCM feature, see the *Global Call API Programming Guide*.

Parameter	Description
<b>target_type</b>	target object type
<b>target_id</b>	target object identifier. This identifier, along with <b>target_type</b> , uniquely specifies the target object.
<b>source_datap</b>	specifies the source data: <ul style="list-style-type: none"> <li>• name - protocol name, board name, and time slot name</li> <li>• identifier - protocol ID, line device</li> <li>• <a href="#">GC_PARM_ID</a> data structure</li> </ul>

Parameter	Description
<b>query_id</b>	specifies which configuration data is required based on the source data: <ul style="list-style-type: none"> <li>GCQUERY_LD_NAME_TO_ID</li> <li>GCQUERY_BOARD_NAME_TO_STATUS</li> <li>GCQUERY_PROTOCOL_NAME_TO_ID</li> <li>GCQUERY_PARM_NAME_TO_ID</li> </ul>
<b>response_datap</b>	specifies the queried result data: <ul style="list-style-type: none"> <li>name - protocol name, board name, and time slot name</li> <li>identifier - protocol ID, line device ID</li> <li><a href="#">GC_PARM_ID</a> data structure</li> </ul>

The **query\_id** parameter specifies which configuration data is required and the data type for the source data (specified in **source\_datap**) as well as the response data (specified in **response\_datap**). The **query\_id** also determines the target object type (see Table 10). The source data pointer, **source\_datap**, points to the location of the known data. The response data parameter is returned by the function. The data type can be a character string, integer, or Global Call parameter set.

To find the query ID, consult the technology notes, release notes, or appropriate header files (for example, *gccfgparam.h*). Table 10 shows the query IDs defined in GCLib and how they are used to obtain configuration information.

**Table 10. Query IDs Defined in GCLib**

Query ID	Source Data Type	Response Data Type	Target Object Type	Explanation
GCQUERY_LD_NAME_TO_ID	string	long	GCTGT_GCLIB_SYSTEM	Find linedev ID by its name
GCQUERY_BOARD_NAME_TO_STATUS	string	int	GCTGT_GCLIB_SYSTEM	Find network interface board status by its name
GCQUERY_PROTOCOL_NAME_TO_ID	string	long	GCTGT_GCLIB_SYSTEM	Find protocol ID by its name. Typically, the protocol name passed in the <a href="#">gc_OpenEx( )</a> function is used.
† Query ID depends on the CCLib implementation.				

Table 10. Query IDs Defined in GCLib (Continued)

Query ID	Source Data Type	Response Data Type	Target Object Type	Explanation
GCQUERY_PARM_NAME_TO_ID†	string	GC_PARM_ID	GCTGT_GCLIB_SYSTEM GCTGT_GCLIB_NETIF GCTGT_GCLIB_CHAN GCTGT_PROTOCOL_SYSTEM GCTGT_PROTOCOL_NETIF GCTGT_PROTOCOL_CHAN	Find set ID, parameter ID by its name. Typically, a CDP parameter name is used. See the <i>Global Call Country Dependent Parameters (CDP) Reference</i> for more information.
† Query ID depends on the CCLib implementation.				

## ■ Termination Events

None

## ■ Cautions

If this function fails, this means that the queried data is not available, and the returned result data pointer may be NULL and cannot be used.

## ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

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

/* query the configuration of a target object */

int query_config(int target_type, int target_id, GC_PARM * source_datap, long query_id,
                GC_PARM * dest_datap)
{
    GC_INFO    t_gcinfo;
    /* call gc_QueryConfigData() function to query the configuration */
    int result = gc_QueryConfigData(target_type, target_id, source_datap, query_id, dest_datap);
    if (result != GC_SUCCESS)
    {
        /* process error by calling gc_ErrorInfo */
        if (gc_ErrorInfo(&t_gcinfo) == GC_SUCCESS)
        {
            printf("Error on target type: 0x%x, target ID: 0x%x\n", target_type, target_id);
            printf("with GC Error 0x%xh: %s\n", t_gcinfo.gcValue, t_gcinfo.gcMsg);
            printf("CCLib %d(%s) Error - 0x%xh: %s\n", t_gcinfo.ccLibId,
                t_gcinfo.ccLibName, t_gcinfo.ccValue, t_gcinfo.ccMsg);
        }
    }
}
```

```

        printf("Additional message: %s\n", t_gcinfo.additionalInfo);
    }
    else
    {
        printf("gc_ErrorInfo() failure");
    }
}
return result;
}

int main()
{
    /* Assume the protocol has been successfully loaded */

    GC_PARM                source_data, dest_data;
    char protocol_name[] = "pdk_ar_r2_io";
    char cdp_name[] = "CDP_SEIZEACK_TIMEOUT";
    char psl_name[] = "SYS_FEATURES";
    GC_PARM_ID             parm_id_stru;
    long                   protocol_id;
    int                    result;

    /* first find the protocol ID by its name */
    source_data.paddress = protocol_name;
    dest_data.longvalue = 0;
    /* call the query_config to find the protocol ID */
    result = query_config(GCTGT_GCLIB_SYSTEM, 0, &source_data,
                        GCQUERY_PROTOCOL_NAME_TO_ID, &dest_data);
    if (result)
    {
        /* can not find the protocol ID */
        return (-1);
    }
    /* found the protocol ID */
    protocol_id = dest_data.longvalue;

    /* then find the parm ID by the CDP name from the protocol */
    source_data.paddress = cdp_name;
    dest_data.pstruct = &parm_id_stru;
    /* call the query_config to find CDP parameter ID */
    result = query_config(GCTGT_PROTOCOL_SYSTEM, protocol_id, &source_data,
                        GCQUERY_PARM_NAME_TO_ID, &dest_data);
    if (!result)
    {
        /* Found the CDP parameter ID*/
        printf("Found the setID: 0x%lx, parmID: 0x%lx, valuetype: %d for CDP parameter: %s\n", parm_id_stru.set_ID, parm_id_stru.parm_ID,
            parm_id_stru.value_type, cdp_name);
    }

    /* find the parm ID by the PSL name from the protocol */
    source_data.paddress = psl_name;
    /* call the query_config to find PSL parameter ID */
    result = query_config(GCTGT_PROTOCOL_SYSTEM, protocol_id, &source_data,
                        GCQUERY_PARM_NAME_TO_ID, &dest_data);
    if (!result)
    {
        /* Found the PSL parameter ID */
        printf("Found the setID: 0x%lx, parmID: 0x%lx, valuetype: %d for PSL parameter: %s\n", parm_id_stru.set_ID, parm_id_stru.parm_ID,
            parm_id_stru.value_type, psl_name);
    }
    return (0);
}

```



■ **See Also**

- [gc\\_GetConfigData\(\)](#)
- [gc\\_SetConfigData\(\)](#)

## gc\_ReleaseCall()

**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 (deprecated)

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN, Analog

DM3: T-1/E-1, ISDN, Analog  
SS7  
IP (embedded stack)

---

### ■ Description

**Note:** The **gc\_ReleaseCall()** function is deprecated in this software release. The **gc\_ReleaseCallEx()** function replaces **gc\_ReleaseCall()** and must be used instead.

The **gc\_ReleaseCall()** function releases the call and the associated internal resources. This function must be called after a **gc\_DropCall()** function completes. The **gc\_ReleaseCall()** function changes the call state from Idle to Null.

## gc\_ReleaseCallEx()

**Name:** int gc\_ReleaseCallEx(crn, mode)

**Inputs:** CRN crn                                      • call reference number  
          unsigned long mode                        • async or sync

**Returns:** 0 if successful  
          <0 if failure.

**Includes:** gcerr.h

**Category:** basic

**Mode:** asynchronous or synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN†, Analog (PDKRT only)†

DM3: T-1/E-1, ISDN†, Analog†  
SS7

IP (host-based stack† and embedded stack)

†See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

The **gc\_ReleaseCallEx()** function releases the call and the associated internal resources. Every **gc\_DropCall()** must be followed by **gc\_ReleaseCallEx()**.

For some technologies, an inbound call will be rejected after **gc\_DropCall()** and prior to **gc\_ReleaseCallEx()**. Refer to the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>crn</b>	call reference number for the call
<b>mode</b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

### ■ Termination Events

GCEV\_RELEASECALL

indicates that the function call is successful and the call resources have been released.

GCEV\_RELEASECALL\_FAIL

indicates that the function failed.

### ■ Cautions

None

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function for error information. If the GCEV\_RELEASECALL\_FAIL event is received, use the **gc\_ResultInfo()** function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
#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. The call has been dropped with gc_DropCall()
 */
int release_callEx(CRN crn)
{
    GC_INFO          gc_error_info;    /* GlobalCall error information data */
    /*
     * Release the system resources using gc_ReleaseCallEx().
     */
    if (gc_ReleaseCallEx(crn, EV_ASYNC) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_ReleaseCallEx() on device handle: 0x%lx,
                GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gCMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    /*
     * Once gc_ReleaseCallEx() returns, the system must wait for the
     * GCEV_RELEASECALL event. When this event is received, the
     * system is then ready to generate or accept another call on
     * this line device.
     */
    return (0);
}
```

## ■ See Also

None

## `gc_ReqANI()`

**Name:** `int gc_ReqANI(crn, anibuf, req_type, mode)`

**Inputs:**

<code>CRN crn</code>	• call reference number
<code>char *anibuf</code>	• buffer to store ANI digits
<code>int req_type</code>	• request type
<code>unsigned long mode</code>	• async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** interface specific

**Mode:** asynchronous or synchronous

**Platform and** Springware: ISDN†

**Technology:** †See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

The **`gc_ReqANI()`** function requests the remote side to return Automatic Number Identification (ANI), 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 if the ANI-on-Demand service is enabled. The returned caller ID (a NULL terminated ASCII string) is stored in the buffer indicated by the **`anibuf`** parameter.

If ANI information is always available, use the **`gc_GetANI()`** function, instead of the **`gc_ReqANI()`** function to obtain a faster return.

See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b><code>crn</code></b>	call reference number
<b><code>anibuf</code></b>	address of the buffer where ANI digits will be loaded. This buffer will be terminated by “\0”.
<b><code>req_type</code></b>	request type: <ul style="list-style-type: none"> <li>• ISDN_CPN_PREF – calling party number preferred</li> <li>• ISDN_BN_PREF – billing number preferred</li> <li>• ISDN_CPN – calling party number only</li> <li>• ISDN_BN – billing number only</li> <li>• ISDN_CA_TSC – special use</li> </ul>
<b><code>mode</code></b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

## ■ Termination Events

### GCEV\_REQANI

indicates successful completion of the function, that is, the requested ANI information was returned.

### GCEV\_TASKFAIL

indicates that the function failed. See the “Error Handling” section in the *Global Call API Programming Guide*.

**Note:** A GCEV\_DISCONNECTED event is an unsolicited event that may be reported to the application after a **gc\_ReqANI()** function is issued.

## ■ Cautions

- Ensure that **anibuf** buffer is at least as large as GC\_ADDRSIZE bytes. Currently, ANI-on-Demand is available only on the AT&T ISDN network.
- If this function is invoked for an unsupported technology, the function will fail. The error value EGC\_UNSUPPORTED will be the Global Call value returned when the **gc\_ErrorInfo()** function is used to retrieve the error code.
- The **gc\_ReqANI()** function may not be supported in all service-provider environments. Check whether retrieving billing information is an available option from your service provider.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function for error information. If the GCEV\_TASKFAIL event is received, use the **gc\_ResultInfo()** function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ 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 <stdio.h>
#include <srllib.h>
#include <gcclib.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_crn(CRN crn, char *ani_buf, int req_type, unsigned long mode)
{
    LINEDEV ldev;          /* Line device */
    GC_INFO gc_error_info; /* GlobalCall error information data */

    if(gc_CRN2LineDev(crn, &ldev) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_CRN2LineDev() on crn: 0x%x, GC ErrorValue: 0x%x - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
                crn, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    if(gc_ReqANI(crn, ani_buf, req_type, mode) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_ReqANI() on device handle: 0x%x, GC ErrorValue: 0x%x - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
                ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    return(0);
}

```

#### ■ See Also

- [gc\\_GetCallInfo\(\)](#)
- [gc\\_GetANI\(\)](#) (deprecated)
- [gc\\_WaitCall\(\)](#)

## **gc\_ReqMoreInfo( )**

**Name:** int gc\_ReqMoreInfo(crn, info\_id, info\_len, timeout, mode)

**Inputs:**

CRN crn	• call reference number
int info_id	• ID of the type of information
int info_len	• amount of information to be collected
int timeout	• time-out value in seconds
unsigned long mode	• async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** optional call handling functions

**Mode:** asynchronous or synchronous

**Platform and Technology:** Springware: T-1/E-1 (PDKRT only), ISDN

DM3: ISDN†  
SS7

†See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

The **gc\_ReqMoreInfo( )** function requests more information such as ANI or DNIS from the call control layer or remote side. The **gc\_ReqMoreInfo( )** function is used after a call to the **gc\_CallAck( )** function, which acknowledges the call and gets the first piece of information.

After an incoming call is received, the application examines the completeness of the destination address. If more information is required (such as DNIS or ANI), the application calls the **gc\_CallAck( )** function with the service type as GCACK\_SERVICE\_INFO to acknowledge the call and request more information. If the technology call control layer does not have the required information or does not have sufficient information, then the application requests additional information from the remote side. If the required information is received, the GCEV\_MOREINFO event is generated. If more digits are still needed, the application then calls the **gc\_ReqMoreInfo( )** function with the number of additional digits to be collected. When all the information is received or when the timer for receiving additional information times out, the GCEV\_MOREINFO event is generated with the appropriate result.

The collected digits can be retrieved by calling the **gc\_GetCallInfo( )** function. When all the information has been collected, the application may call the **gc\_CallAck( )** function with the service type as GCACK\_SERVICE\_PROC to indicate to the remote side that the call is proceeding.

In synchronous mode, the **gc\_ReqMoreInfo( )** function will return successfully regardless of the status of the requested information, that is, if all, some or none of the information is available. If the call control technology is still waiting for information, then the function blocks until the



information is received or the **timeout** expires. The [gc\\_GetCallInfo\(\)](#) function must be called to determine if any information was received. If an error occurs, the function returns -1.

For more information about the use of the **gc\_ReqMoreInfo()** function, see the discussion of overlap receiving in the *Global Call API Programming Guide*.

Parameter	Description
<b>crn</b>	call reference number for the call
<b>info_id</b>	ID of the type of information to be collected. The valid values are: <ul style="list-style-type: none"> <li>• <b>DESTINATION_ADDRESS</b> – Request the called party number (typically digits - DNIS).</li> <li>• <b>ORIGINATION_ADDRESS</b> – Request the calling party number (typically digits - ANI).</li> </ul>
<b>info_len</b>	amount of the information to be collected
<b>timeout</b>	specifies the amount of time in seconds in which the additional information must be generated. The <b>GCEV_MOREINFO</b> event is generated with the appropriate result value when the timer expires. The time-out value must be a non-zero positive value. A value $\leq 0$ means that the function will wait indefinitely until the additional digits are received.
<b>mode</b>	set to <b>EV_ASYNC</b> for asynchronous execution or to <b>EV_SYNC</b> for synchronous execution

## ■ Termination Events

### GCEV\_MOREINFO

indicates that the function call is successful and the requested digits have been received. The **eventdatap** field of the **METAEVENT** structure contains the status of the information. Use the [gc\\_ResultValue\(\)](#) function to get the result value that indicates if the requested number of digits is available, more digits are available, or no digits are available. Table 11 indicates the valid values.

### GCEV\_TASKFAIL

indicates that the function failed.

**Table 11. Result Values for GCEV\_MOREINFO**

Result Value	Description
GCRV_INFO_PRESENT_ALL	The requested digits are now available
GCRV_INFO_PRESENT_MORE	The requested digits are now available. More/additional digits are available
GCRV_INFO_SOME_TIMEOUT	Only some digits are available - timed out
GCRV_INFO_SOME_NOMORE	Only some digits are available, no more digits will be received
GCRV_INFO_NONE_TIMEOUT	No digits are available - timed out
GCRV_INFO_NONE_NOMORE	No more digits are available

## ■ Cautions

The **gc\_ReqMoreInfo( )** function can only be called after the **gc\_CallAck( )** function is called.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
/*
 * 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
 * 5. It is determined that more digits are required. The call is
 *    acknowledged with gc_callAck() with DESTINATION_ADDRESS
 *    parameter to request more DNIS
 * 6. The GCEV_MOREINFO event is received.
 */
int req_moreinfo(CRN crn)
{
    GC_INFO    gc_error_info; /* GlobalCall error information data */
    int        n_digits;      /* to be evaluated */

    /*
     * Do the following:
     * 1. Get called party number using gc_GetCallInfo() and evaluate it.
     * 2. If three more digits are required by application to properly
     *    process or route the call, request that they be sent
     *    => n_digits = 3.
     */

    if(gc_ReqMoreInfo(crn, DESTINATION_ADDRESS, n_digits, 0, EV_ASYNC) != GC_SUCCESS)
    {
        /* process error return as shown */
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_ReqMoreInfo() on device handle: 0x%lx,
                GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    return (0);
}
/*
 * The application must wait for the GCEV_MOREINFO event.
 * The cause of the event will indicate if all the digits have
 * been received. If digits have been received, then collect
 * the remaining digits by calling gc_GetCallInfo()
 */
```

```
*  
*  
* Application can then answer, accept, or terminate the call at this  
* point, based on the DNIS information.  
*/
```

#### ■ See Also

- [gc\\_GetCallInfo\( \)](#)
- [gc\\_CallAck\( \)](#)
- [gc\\_SendMoreInfo\( \)](#)

## gc\_ReqService( )

**Name:** int gc\_ReqService(target\_type, target\_ID, pserviceID, reqdatap, respdatap, mode)

**Inputs:**

int target_type	• type of target object
long target_ID	• ID of target object
unsigned long *pserviceID	• pointer to service ID
GC_PARM_BLK reqdatap	• pointer to data associated with the request
GC_PARM_BLK *respdatap	• pointer to location that stores the pointer to the response buffer generated by the call control library
unsigned long mode	• async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** Global Call Service Request (GCSR)

**Mode:** asynchronous or synchronous

**Platform and Technology:** IP (host-based stack)†

†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The **gc\_ReqService( )** function requests a service from a remote device. Some examples of requests are device registration, channel setup, call setup, and information request. See the appropriate *Global Call Technology User's Guide* for the service requests supported under each technology.

For more information about the **gc\_ReqService( )** function, see the discussion of the Global Call Service Request (GCSR) feature in the *Global Call API Programming Guide*.

Parameter	Description
<b>target_type</b>	target object type. Valid values are: <ul style="list-style-type: none"> <li>• GCTGT_GCLIB_CHAN</li> <li>• GCTGT_GCLIB_CRN</li> </ul>
<b>target_id</b>	target object identifier. This identifier, along with <b>target_type</b> , uniquely specifies the target object. Valid identifiers are: <ul style="list-style-type: none"> <li>• line device handle</li> <li>• call reference number</li> </ul>
<b>pserviceID</b>	points to the service ID of this request. Assigned by the call control library when this function returns.
<b>reqdatap</b>	points to the user-specified data associated with the request

Parameter	Description
<b>respdatap</b>	points to the address of the response buffer ( <a href="#">GC_PARM_BLK</a> ) generated by the call control library. See the appropriate <i>Global Call Technology User's Guide</i> on how to set up the data structure for a particular technology. Set to NULL in asynchronous mode.  If a <a href="#">GC_PARM_BLK</a> buffer is returned, the parameters in this buffer must be processed or copied prior to the next Global Call function call. The reason for this is that the <a href="#">GC_PARM_BLK</a> buffer will be deallocated in a subsequent Global Call function call.
<b>mode</b>	set to <code>EV_ASYNC</code> for asynchronous execution or to <code>EV_SYNC</code> for synchronous execution

### ■ Termination Events

`GCEV_SERVICERESP`

indicates a response to a request for a service from a remote device.

`GCEV_TASKFAIL`

indicates that the function failed.

**Note:** The `extevtdatap` field of these events contains a pointer to a [GC\\_PARM\\_BLK](#) that contains an unsigned long value that is the Service ID associated with the event. This pointer is only valid until the next [gc\\_GetMetaEvent\(\)](#) or [gc\\_GetMetaEventEx\(\)](#) is called. See the appropriate *Global Call Technology User's Guide* for technology-specific information.

### ■ Cautions

- In synchronous mode, if the user specifies that no response is necessary, no data will be stored in the **respdatap** structure.
- Only synchronous mode is supported for the following target objects: `GCTGT_GCLIB_SYSTEM`, `GCTGT_CCLIB_SYSTEM`, `GCTGT_PROTOCOL_SYSTEM`, and `GCTGT_FIRMWARE_NETIF`. Otherwise, the function will return the async mode error.
- The [GC\\_PARM\\_BLK](#) buffer, if one is returned via the **respdatap** pointer, has a guaranteed persistence only until the next Global Call function call. Any parameters within this buffer must be processed or copied within the application, or risk being lost when the next Global Call function call is invoked.
- When using the `gc_ReqService()` function, `PARM_REQTYPE` and `PARM_ACK` are mandatory parameters of the [GC\\_PARM\\_BLK](#) pointed to by the **reqdatap** function parameter.

### ■ Errors

If this function returns `<0` to indicate failure, use the `gc_ErrorInfo()` function for error information. If the `GCEV_TASKFAIL` event is received, use the `gc_ResultInfo()` function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdner.h` file for the ISDN call control library).

**■ Example**

```

#include "gclib.h"

void main( )
{
    LINEDEV      devh;
    unsigned long serviceID;
    GC_PARM_BLK  reqdatap = NULL;
    GC_INFO      gc_error_info;  /* GlobalCall error information data */

    /* Following code assumes that gc_OpenEx has been done with handle = devh */

    /* Set up GC_PARM_BLK */
    if ( gc_util_insert_parm_val( &reqdatap, GCSET_SERVREQ, PARM_REQTYPE,
        sizeof( int ), 0 ) != GC_SUCCESS )
    {
        /* Process error */
    }
    if ( gc_util_insert_parm_val( &reqdatap, GCSET_SERVREQ, PARM_ACK,
        sizeof( short ), GC_NACK ) != GC_SUCCESS )
    {
        /* Process error */
    }

    /* Insert any other technology-dependent parameters */

    if ( gc_ReqService( GCTGT_GCLIB_CHAN, devh, &serviceID, reqdatap, NULL,
        EV_ASYNC ) != GC_SUCCESS )
    {
        /* Process error */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_ReqService() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
            CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
            devh, gc_error_info.gcValue, gc_error_info.gcMsg,
            gc_error_info.ccLibId, gc_error_info.ccLibName,
            gc_error_info.ccValue, gc_error_info.ccMsg);
    }

    /* Wait for GCEV_SERVRESP and process */

    /* Delete GC_PARM_BLK */
    gc_util_delete_parm_blk( reqdatap );
}

```

**■ See Also**

- [gc\\_RespService\(\)](#)

## `gc_ResetLineDev()`

**Name:** `int gc_ResetLineDev(linedev, mode)`

**Inputs:** `LINEDEV linedev` • Global Call line device handle  
`unsigned long mode` • async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** system controls and tools

**Mode:** asynchronous or synchronous

**Platform and** All†

**Technology:** †See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The **`gc_ResetLineDev()`** function resets the line device state and disconnects calls on the line device. All calls being set up are aborted. This function typically is used after a recovery from a trunk error, a recovery from an alarm condition, or to reset 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.

See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b><code>linedev</code></b>	Global Call line device
<b><code>mode</code></b>	set to <code>EV_ASYNC</code> for asynchronous execution or to <code>EV_SYNC</code> for synchronous execution

### ■ Termination Events

`GCEV_RESETLINEDEV`  
indicates that the function was successful, that is, the line device was reset to the Null state.

`GCEV_RESTARTFAIL`  
indicates that the function failed. See the “Error Handling” section in the *Global Call API Programming Guide*.

`GCEV_TASKFAIL`  
indicates that the function failed.

**Note:** See the appropriate *Global Call Technology User's Guide* for technology-specific information.

## ■ Cautions

- When a time slot (for example, dtiB1T1) is opened, it is put in the out-of-service state, blocking all incoming calls on that time slot. The application must issue a [gc\\_WaitCall\(\)](#) to accept incoming calls or a [gc\\_MakeCall\(\)](#) to make outgoing calls. When [gc\\_ResetLineDev\(\)](#) is issued, it puts the time slot in the out-of-service state, disconnecting any existing calls and blocking any further incoming calls.

Any synchronous call issued on this time slot—for example, [gc\\_WaitCall\(\)](#)—is aborted. The [gc\\_ResetLineDev\(\)](#) function does not terminate a synchronous call issued in a different process.

- The [gc\\_ResetLineDev\(\)](#) function should be used when the application has lost control of the line due to a protocol error. It should not be used to switch between states, such as to simplify the call control process. In general, calling [gc\\_ResetLineDev\(\)](#) from the Idle state is an acceptable practice, but using [gc\\_ResetLineDev\(\)](#) in some call states can lead to other unnecessary protocol errors.
- After successful completion of this function, the application must issue a new [gc\\_WaitCall\(\)](#) function to return the channel to the Idle state to be ready to receive the next call on the channel.
- Do not call any Global Call function until after the GCEV\_RESETLINEDEV event is received for this line. Likewise, ignore any other events until after the GCEV\_RESETLINEDEV event is received.

## ■ Errors

If this function returns <0 to indicate failure, use the [gc\\_ErrorInfo\(\)](#) function for error information. If the GCEV\_RESTARTFAIL or GCEV\_TASKFAIL event is received, use the [gc\\_ResultInfo\(\)](#) function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <gcilib.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 */
    GC_INFO gc_error_info;    /* GlobalCall error information data */

    /*
     * 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 */
            gc_ErrorInfo( &gc_error_info );
            printf ("Error: gc_ResetLineDev() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
                    CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                    port[i].ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                    gc_error_info.ccLibId, gc_error_info.ccLibName,
                    gc_error_info.ccValue, gc_error_info.ccMsg);
            return (gc_error_info.gcValue);
        }

        /*
         * Application will need to re-issue gc_WaitCall() to wait
         * for incoming calls
         */
    }
    return (0);
}

```

## ■ See Also

- [gc\\_WaitCall\(\)](#)

## gc\_RespService( )

**Name:** int gc\_RespService(target\_type, target\_ID, datap, mode)

**Inputs:**

int target_type	• type of target object
long target_ID	• ID of target object
GC_PARM_BLK datap	• pointer to data associated with the response
unsigned long mode	• async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** Global Call Service Request (GCSR)

**Mode:** asynchronous or synchronous

**Platform and Technology:** Springware: ISDN BRI†

IP (host-based stack)†

†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The **gc\_RespService( )** function generates a response to a requested service. Notification of a request is indicated by the reception of a GCEV\_SERVICEREQ event. The extevtdatap field of the event contains a pointer to a **GC\_PARM\_BLK** that contains an unsigned long value that is the Service ID associated with the event.

For more information about the **gc\_RespService( )** function, see the discussion of the Global Call Service Request (GCSR) feature in the *Global Call API Programming Guide*.

Parameter	Description
<b>target_type</b>	target object type. Valid values are: <ul style="list-style-type: none"> <li>• GCTGT_GCLIB_CHAN</li> <li>• GCTGT_GCLIB_CRN</li> </ul>
<b>target_id</b>	target object identifier. This identifier, along with <b>target_type</b> , uniquely specifies the target object. Valid identifiers are: <ul style="list-style-type: none"> <li>• line device handle</li> <li>• call reference number</li> </ul>
<b>datap</b>	points to data associated with the response; contains user-specified values
<b>mode</b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

## ■ Termination Events

GCEV\_SERVICERESPCMPLT

indicates that a response has been successfully generated

GCEV\_TASKFAIL

indicates that the function failed

**Note:** The `extevtdatap` field of these events contains a pointer to a `GC_PARM_BLK` that contains an unsigned long value that is the Service ID associated with the event. This pointer is only valid until the next `gc_GetMetaEvent()` or `gc_GetMetaEventEx()` is called. See the appropriate *Global Call Technology User's Guide* for technology-specific information.

## ■ Cautions

- Only synchronous mode is supported for the following target objects: GCTGT\_GCLIB\_SYSTEM, GCTGT\_CCLIB\_SYSTEM, GCTGT\_PROTOCOL\_SYSTEM, and GCTGT\_FIRMWARE\_NETIF. Otherwise, the function will return the async mode error.
- When using the `gc_RespService()` function, `PARM_SERVICEID` is a mandatory parameter of the `GC_PARM_BLK` pointed to by the `datap` function parameter.

## ■ Errors

If this function returns `<0` to indicate failure, use the `gc_ErrorInfo()` function for error information. If the GCEV\_TASKFAIL event is received, use the `gc_ResultInfo()` function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

```
#include "gclib.h"

void main( )
{
    LINEDEV      devh;
    unsigned long serviceID;
    GC_PARM_BLKP  datap = NULL;
    GC_INFO      gc_error_info;    /* GlobalCall error information data */

    /* Following code assumes that:
     * 1. gc_OpenEx has been done with handle = devh
     * 2. a GCEV_SERVREQ event has been received and processed, with serviceID
     *    set to the appropriate value
     */

    /* Set up GC_PARM_BLK for reply */
    if ( gc_util_insert_parm_ref( &datap, GCSET_SERVREQ, PARM_SERVICEID,
        sizeof( unsigned long ), &serviceID ) != GC_SUCCESS )
    {
        /* Process error */
    }
    if ( gc_util_insert_parm_val( &datap, GCSET_SERVREQ, PARM_REQTYPE,
        sizeof( int ), 0 ) != GC_SUCCESS )
    {
        /* Process error */
    }
}
```

```
if ( gc_util_insert_parm_val( &datap, GCSET_SERVREQ, PARM_ACK,
    sizeof( short ), GC_ACK ) != GC_SUCCESS )
{
    /* Process error */
}
/* Insert any other technology-dependent parameters */

if ( gc_RespService( GCTGT_GCLIB_CHAN, devh, datap, EV_SYNC ) != GC_SUCCESS )
{
    /* Process error */
    /* Process error */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_RespService() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
        CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
        devh, gc_error_info.gcValue, gc_error_info.gcMsg,
        gc_error_info.ccLibId, gc_error_info.ccLibName,
        gc_error_info.ccValue, gc_error_info.ccMsg);
}

/* Delete GC_PARM_BLK */
gc_util_delete_parm_blk( datap );
}
```

■ **See Also**

- [gc\\_ReqService\( \)](#)

## gc\_ResultInfo()

**Name:** int gc\_ResultInfo(\*a\_Metaevent, a\_Info)

**Inputs:** METAEVENT \*a\_Metaevent      • pointer to the structure of metaevent data  
GC\_INFO\* a\_Info                      • pointer to the GC\_INFO data structure

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

**Includes:** gcLib.h  
gcerr.h

**Category:** system controls and tools

**Mode:** synchronous

**Platform and Technology:** All

---

### ■ Description

The **gc\_ResultInfo()** function retrieves information about Global Call events. Calling the **gc\_ResultInfo()** function is equivalent to calling the **gc\_ResultValue()**, **gc\_ResultMsg()**, and **gc\_CCLibNameToID()** functions in turn. To retrieve the information, the **gc\_ResultInfo()** function must be called immediately after a Global Call event arrives and before the next event is requested.

If the application gets an event, the gcValue or ccValue field of the **GC\_INFO** structure identifies the cause of the event to which the **a\_Metaevent** parameter points. This pointer is acquired by the **gc\_GetMetaEvent()** or **gc\_GetMetaEventEx()** function.

Parameter	Description
<b>a_Metaevent</b>	points to the metaevent associated with this event
<b>a_Info</b>	points to the GC_INFO structure where information about the event is contained. See <b>GC_INFO</b> , on page 388 for more information.

### ■ Termination Events

None

### ■ Cautions

- The **gc\_ResultInfo()** function can be called only in the **same** thread in which the event was retrieved and before the next Global Call event is received.
- Do not overwrite the message space pointed to by any of the char\* fields in **GC\_INFO** as these point to private internal space.
- The lifetime of the strings pointed to by the **GC\_INFO** data structure is from the time the **gc\_ResultInfo()** function returns to the time the next event is requested.

## ■ Errors

If the **gc\_ResultInfo()** function fails, error analysis should not be done by calling either the **gc\_ErrorInfo()** or **gc\_ErrorValue()** function. A failure return generally indicates that **a\_Info** is NULL.

## ■ Example

```
/*
-- This function can be called anytime after a GlobalCall event has occurred
-- This procedure prints the result information to the console with no other side effects
*/
void PrintResultInfo(METAEVENT *a_metaeventp)
{
    int         retCode;
    GC_INFO     t_Info;
    GC_INFO     gc_error_info; /* GlobalCall error information data */

    retCode = gc_ResultInfo(a_metaeventp, &t_Info);
    if (retCode == GC_SUCCESS) {
        printf("gc_ResultInfo() successfully called\n");
        PrintGC_INFO(&t_Info);
    } else {
        printf("gc_ResultInfo() call failed\n");
    }
}

/*
-- This function is called to print GC_INFO to the system console
-- Typically it would be called after a call to gc_ErrorInfo()
-- or gc_ResultInfo() to print the resulting GC_INFO data structure
*/
void PrintGC_INFO(GC_INFO *a_Info)
{
    printf("a_Info->gcValue = 0x%x\n", a_Info->gcValue);
    printf("a_Info->gcMsg = %s\n", a_Info->gcMsg);
    printf("a_Info->ccLibId = %d\n", a_Info->ccLibId);
    printf("a_Info->ccLibName = %s\n", a_Info->ccLibName);
    printf("a_Info->ccValue = 0x%x\n", a_Info->ccValue);
    printf("a_Info->ccMsg = %s\n", a_Info->ccMsg);
    printf("a_Info->additionalInfo = %s\n", a_Info->additionalInfo);
    printf("Enter <CR> to continue: ");
    getchar();
}
```

## ■ See Also

- [gc\\_ResultValue\(\)](#) (deprecated)
- [gc\\_ResultMsg\(\)](#) (deprecated)
- [gc\\_CCLibIDToName\(\)](#)

## `gc_ResultMsg()`

**Name:** `int gc_ResultMsg(cclibid, result_code, msg)`

**Inputs:**

<code>int cclibid</code>	• call control library ID
<code>long result_code</code>	• used to get associated message
<code>char **msg</code>	• pointer to address of returned message string

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** system controls and tools (deprecated)

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN, Analog  
DM3: T-1/E-1, ISDN, Analog  
SS7  
IP (embedded stack)

---

### ■ Description

**Note:** The `gc_ResultMsg()` function is deprecated in this software release. The suggested alternative is `gc_ResultInfo()` to retrieve event information or `gc_ErrorInfo()` to retrieve error information.

The `gc_ResultMsg()` function retrieves an ASCII string describing a 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
<b>cclibid</b>	call control library identification from which the <b>result_code</b> was generated. If the <b>result_code</b> value is a Global Call error code or result value, then set <b>cclibid</b> to <code>LIBID_GC</code> .
<b>result_code</b>	result value of the event or error code from the library, <b>cclibid</b>
<b>msg</b>	points to address where the description of the <b>result_code</b> message will be stored

### ■ Termination Events

None

### ■ Cautions

Do not overwrite the **\*msg** pointer since it points to private internal Global Call data space.

## ■ Errors

If this function returns <0 to indicate failure, no additional information is available to indicate why the function failed. Generally, a failure of this function is caused by passing an invalid **cclibid** or **msg** parameter.

## ■ Example

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

/* cclibid = LIBID_GC to print GC Lib's result value */
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("%s library had error 0x%lx - %s\n", lib_name, result_code, msg);
    } else {
        printf("gc_ResultMsg failed\n");
    }
}
```

## ■ See Also

- [gc\\_ResultInfo\(\)](#)
- [gc\\_ResultValue\(\)](#) (deprecated)



## 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 Global Call 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:** gcplib.h  
gcerr.h

**Category:** system controls and tools

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN, Analog

DM3: T-1/E-1, ISDN, Analog

SS7†

IP (embedded stack)

†See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

**Note:** The **gc\_ResultValue()** function is deprecated in this software release. The suggested equivalent is **gc\_ResultInfo()**.

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 Global Call 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
<b>metaeventp</b>	points to the event data block. This pointer is acquired via the <a href="#">gc_GetMetaEvent()</a> function or, for the Windows extended asynchronous model only, the <a href="#">gc_GetMetaEventEx()</a> function.
<b>gc_resultp</b>	address where the Global Call result value is to be stored
<b>cclibidp</b>	address where the identification of the call control library associated with this metaevent is to be stored
<b>cclib_resultp</b>	address where the result value associated with the call control library metaevent is to be stored

### ■ Termination Events

None

## ■ Cautions

None

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnnerr.h* file for the ISDN call control library).

## ■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <gcclib.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 */
    char         *lib_name;     /* pointer to library name for cclibid */

    /* Obtain the event data */
    sr_waitevt(-1);              /* Wait indefinitely for an event */
    if (gc_GetMetaEvent(&metaevent) != GC_SUCCESS) {
        /* Process error return as shown in the
        Example code of the gc_ErrorValue()
        function description */

        gc_GetMetaEvent(&metaevent); /* Get event parameters into metaevent */
        if (metaevent.flags & GCME_GC_EVENT) { /* Process GlobalCall events */
            /* 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("GlobalCall event 0x%x received on LDEV: %ld - %s\n",
                    metaevent.evtttype, metaevent.evtdev, msg);
                gc_CCLibIDToName(cclibid, &lib_name);
                gc_ResultMsg(cclibid, cc_result, &msg);
                printf("%s 0x%x received on LDEV: %ld - %s\n", lib_name,
                    metaevent.evtttype, metaevent.evtdev, msg);
                return(0);
            }
        }
        else {
            printf("could not get result value\n");
            return (-1); /* indicate error */
        }
    }
}
```

## ■ See Also

- [gc\\_ResultInfo\(\)](#)
- [gc\\_ResultMsg\(\)](#) (deprecated)
- [gc\\_CCLibIDToName\(\)](#)

## `gc_RetrieveAck()`

**Name:** `int gc_RetrieveAck(crn)`

**Inputs:** CRN `crn` • call reference number

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** interface specific

**Mode:** synchronous

**Platform and Technology:** Springware: ISDN BRI, ISDN PRI (DPNSS and Q.SIG only)

---

### ■ Description

The `gc_RetrieveAck()` function accepts a retrieve request from remote equipment. The call must be in OnHold state and the GCEV\_RETRIEVECALL event must be received before the function is called.

Parameter	Description
<code>crn</code>	call reference number

### ■ Termination Events

None

### ■ Cautions

- The ISDN Q.SIG protocol allows the retrieval of a call in the OnHold state to be accepted or rejected. If this function is used with a protocol that does not support the retrieval of a call in the OnHold state, the function fails.
- The `gc_RetrieveAck()` function is called only after the call is in the OnHold state and after the GCEV\_RETRIEVECALL event is received, or the function fails.

### ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

```

/*
 * Assume the following has been done:
 * 1. device has been opened (e.g. :N_dtiBt1:P_isdn,
 *    :N_dtiBt2:P_isdn, etc...)
 * 2. gc_WaitCall() has been issued to wait for a call.
 * 3. gc_GetMetaEvent() or gc_GetMetaEventEx() (Windows) has been
 *    called to convert the event into metaevent.
 * 4. a GCEV_OFFERED event has been detected.
 */

#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>

METAEVENT metaevent;

int retrieve_call()
{
    LINEDEV  ldev;          /* Line device */
    CRN      crn;           /* call reference number */
    GC_INFO  gc_error_info; /* GlobalCall error information data */

    /*
     * Do the following:
     * 1. Get the CRN from the metaevent
     * 2. Proceed to retrieve the call on hold
     */

    crn = metaevent.crn;

    if (gc_RetrieveCall(crn, EV_ASYNC) != GC_SUCCESS) {
        /* Process error */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_RetrieveCall() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    return 0;
}

/* Retrieve events from SRL */
int evt_hdlr()
{
    LINEDEV  ldev;          /* Line device */
    CRN      crn;           /* Call Reference Number */
    GC_INFO  gc_error_info; /* GlobalCall error information data */
    int      retcode;        /* Return Code */

    retcode = gc_GetMetaEvent(&metaevent);
    if (retcode != GC_SUCCESS) {
        /* get and process the error */
    } else {
        /* Continue with normal processing */
    }

    crn = metaevent.crn;

    switch(metaevent.evttype) {

```

```

case GCEV_RETRIEVECALL:
    /* retrieve signaling information from queue */
    if (gc_RetrieveAck(crn) != GC_SUCCESS) {
        /* Process error */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_RetrieveAck() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    else {
        /* succeeded, process signaling information */
    }

    break;
}
return 0;
}

```

#### ■ See Also

- [gc\\_HoldCall\(\)](#)
- [gc\\_RetrieveCall\(\)](#)
- [gc\\_RetrieveRej\(\)](#)

## gc\_RetrieveCall()

**Name:** int gc\_RetrieveCall(crn, mode)

**Inputs:** CRN crn • call reference number  
unsigned long mode • async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** advanced call model

**Mode:** asynchronous or synchronous

**Platform and Technology:** Springware: T-1/E-1 (PDKRT only)†, ISDN BRI, ISDN PRI (DPNSS and Q.SIG only)

**Technology:** SS7†

†See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

The **gc\_RetrieveCall()** function retrieves a call from the OnHold state. The call must be in the OnHold state before calling this function.

Parameter	Description
<b>crn</b>	call reference number
<b>mode</b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

### ■ Termination Events

#### ISDN events:

##### GCEV\_RETRIEVEACK

indicates that the function was successful, that is, the call was retrieved from the OnHold state

##### GCEV\_RETRIEVEREJ

indicates that the function failed, that is, the call was not retrieved from hold

#### PDKRT events:

##### GCEV\_RETRIEVECALL

indicates that the function was successful, that is, the call was retrieved from the OnHold state

##### GCEV\_TASKFAIL

indicates that the function failed. For more information, see the “Error Handling” section in the *Global Call API Programming Guide*.

## ■ Cautions

The **`gc_RetrieveCall()`** function can only be used when the call is in the OnHold state or the function fails.

## ■ Errors

If this function returns <0 to indicate failure, use the **`gc_ErrorInfo()`** function for error information. If the GCEV\_RETRIEVEREJ or GCEV\_TASKFAIL event is received, use the **`gc_ResultInfo()`** function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ 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) has been
 *    called to convert the event into metaevent.
 * 4. a GCEV_OFFERED event has been detected.
 */

#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>

METAEVENT metaevent;

int retrieve_call()
{
    LINEDEV ldev;          /* Line device */
    CRN crn;               /* call reference number */
    GC_INFO gc_error_info; /* GlobalCall error information data */
    /*
     * Do the following:
     * 1. Get the CRN from the metaevent
     * 2. Proceed to retrieve the call on hold
     */
    crn = metaevent.crn;
    if (gc_RetrieveCall(crn, EV_ASYNC) != GC_SUCCESS) {
        /* Process error */
        gc_ErrorInfo(&gc_error_info);
        printf ("Error: gc_RetrieveCall() on device handle: 0x%x, GC ErrorValue: 0x%x - %s,\n",
                CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
                metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    return 0;
}
```

*gc\_RetrieveCall()* — retrieve a call from the OnHold state



■ See Also

- [gc\\_HoldCall\(\)](#)
- [gc\\_RetrieveAck\(\)](#)
- [gc\\_RetrieveRej\(\)](#)





- Not all ISDN Network cause/error codes are universally supported across switch types. Before you use a particular cause code, compare its validity with the appropriate switch vendor specifications.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ 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) has been
 *    called to convert the event into metaevent.
 * 4. a GCEV_OFFERED event has been detected.
 */

#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>

METAEVENT metaevent;

int retrieve_call()
{
    LINEDEV  ldev;          /* Line device */
    CRN      crn;           /* call reference number */
    GC_INFO  gc_error_info; /* GlobalCall error information data */

    /*
     * Do the following:
     * 1. Get the CRN from the metaevent
     * 2. Proceed to retrieve the call on hold
     */

    crn = metaevent.crn;

    if(gc_RetrieveCall(crn, EV_ASYNC) != GC_SUCCESS) {
        /* Process error */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_RetrieveCall() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,\n",
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    return 0;
}
```

```

/* Retrieve events from SRL */
int evt_hdlr()
{
    LINEDEV   ldev;           /* Line device */
    CRN       crn;           /* Call Reference Number */
    GC_INFO   gc_error_info; /* GlobalCall error information data */
    int       retcode;       /* Return Code */

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

    crn = metaevent.crn;

    switch(metaevent.evtttype) {

case GCEV_RETRIEVECALL:
    /* retrieve signaling information from queue */
    if (gc_RetrieveRej(crn, TEMPORARY_FAILURE) != GC_SUCCESS) {
        /* Process error */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_RetrieveRej() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    else {
        /* succeeded, process signaling information */
    }

    break;
    }
    return 0;
}

```

#### ■ See Also

- [gc\\_HoldCall\(\)](#)
- [gc\\_RetrieveAck\(\)](#)
- [gc\\_RetrieveCall\(\)](#)

## **gc\_SendMoreInfo( )**

**Name:** int gc\_SendMoreInfo(crn, info\_id, info\_ptr, mode)

**Inputs:**

CRN crn	• call reference number
int info_id	• ID of the type of information requested
GC_PARM *info_ptr	• pointer to information to be sent
unsigned long mode	• async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** optional call handling functions

**Mode:** asynchronous or synchronous

**Platform and Technology:** Springware: T-1/E-1 (PDKRT only), ISDN

DM3: ISDN†  
SS7

†See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

The **gc\_SendMoreInfo( )** function sends more information to the remote side (such as digits). This function allows the application to initiate the sending of information such as in overlap sending (for example, a gateway application).

The **gc\_SendMoreInfo( )** function can be called in the Dialing state after **gc\_MakeCall( )** is issued or after receiving a GCEV\_REQMOREINFO event. This event is received if the outgoing call does not contain sufficient information and the technology call control layer or the remote side requests more information. The result value of this event indicates what type of information was requested (either destination or origination address). The **gc\_SendMoreInfo( )** function can be called multiple times to send more information regardless of whether or not the GCEV\_REQMOREINFO event was received. If information is requested and there is no more information left to send, the **gc\_SendMoreInfo( )** can be called again with the **info\_ptr** pointer set to NULL. When sufficient information has been sent, a GCEV\_PROCEEDING, GCEV\_ALERTING, or GCEV\_CONNECTED event should be received.

Although **gc\_SendMoreInfo( )** can also be called prior to receiving a GCEV\_REQMOREINFO event (for example, in the Dialing state), there is a possibility of a glare condition where the **gc\_SendMoreInfo( )** function is called and then a GCEV\_REQMOREINFO event is received.

For more information about the **gc\_SendMoreInfo( )** function, see the discussion of overlap sending in the *Global Call API Programming Guide*.

Parameter	Description
<b>crn</b>	call reference number for the call
<b>info_id</b>	ID of the type of information to be sent. The valid values are: <ul style="list-style-type: none"> <li>• <code>DESTINATION_ADDRESS</code> – send the called party number (DNIS)</li> <li>• <code>ORIGINATION_ADDRESS</code> – send the calling party number (ANI)</li> </ul>
<b>info_ptr</b>	points to information to be sent. If no information is to be sent, this parameter must be set to <code>NULL</code> .
<b>mode</b>	set to <code>EV_ASYNC</code> for asynchronous execution or to <code>EV_SYNC</code> for synchronous execution

## ■ Termination Events

### GCEV\_SENDMOREINFO

indicates that all the digits are successfully sent

### GCEV\_TASKFAIL

indicates that the function failed and that the request/message was rejected by the protocol or firmware

## ■ Cautions

- Do not resend data already sent. Send only additional information or `NULL` if there is no more information to send.
- Always respond to the `gc_ReqMoreInfo()` function by calling the `gc_SendMoreInfo()` function to pass initial data or `NULL` if there is no more data to send.

## ■ Errors

If this function returns `<0` to indicate failure, use the `gc_ErrorInfo()` function for error information. If the `GCEV_TASKFAIL` event is received, use the `gc_ResultInfo()` function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnnerr.h` file for the ISDN call control library).

## ■ Example

```
/*
 * Assume the following has been done:
 * 1. Opened line devices for each time slot on DTIB1.
 * 2. Make a call using gc_MakeCall()
 * 3. The call is in Dialing state.
 * 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_REQMOREINFO event
 */
int send_moreinfo(CRN crn, char *infop)
{
    GC_INFO gc_error_info; /* GlobalCall error information data */
```

```
/*
 * Setup the GC_PARM structure
 */
GC_PARM info;
info.paddress = infop;

if (gc_SendMoreInfo(crn, DESTINATION_ADDRESS, &info, EV_ASYNC)
    != GC_SUCCESS)
{
    /* process error return as shown */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_SendMoreInfo() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
           CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
           metaevent.evtdev, gc_error_info.gcValue, gc_error_info.gcMsg,
           gc_error_info.ccLibId, gc_error_info.ccLibName,
           gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}
return (0);
}

/*
 * The application may get the GCEV_SENDMOREINFO termination event.
 *
 * The remote side will then answer, accept, or terminate the call at this
 * point.
 */
```

#### ■ See Also

- [gc\\_ReqMoreInfo\(\)](#)

## `gc_SetAlarmConfiguration()`

**Name:** `int gc_SetAlarmConfiguration(linedev, aso_id, alarm_list, alarm_config_type)`

**Inputs:**

<code>LINEDEV linedev</code>	• line device ID
<code>unsigned long aso_id</code>	• alarm source object ID
<code>ALARM_LIST *alarm_list</code>	• pointer to alarm list
<code>int alarm_config_type</code>	• alarm information type

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** GCAMS

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN

DM3: T-1/E-1, ISDN  
IP (host-based stack)

---

### ■ Description

The `gc_SetAlarmConfiguration()` function sets alarm configuration parameter values for alarms originating from the specified alarm source object.

The function can be used as follows:

- To specify whether the application wants to be notified when any of the alarms in the **alarm\_list** occur. The default is not to notify the application of the alarms.
- To classify the alarms in **alarm\_list** as either blocking or non-blocking. The default for the severity of the alarm, that is, whether the alarm is blocking or non-blocking, is ASO dependent. For a list of alarms for the ASOs that are maintained by Global Call, see [Section 6.1, “Alarm Source Object IDs”](#), on page 417. For a list of technology-specific alarms, see the appropriate *Global Call Technology User’s Guide*.

The `gc_SetAlarmConfiguration()` function can also be used to determine if an alarm exists on a line device when the device is opened. After calling `gc_OpenEx()`, enable alarm notification for the line device. If any alarms are already active on the board, the application will be notified.

The list of alarms to be configured is stored in the [ALARM\\_LIST](#) data structure. (See [ALARM\\_LIST](#), on page 368.) For each alarm, the `alarm_number` field in the [ALARM\\_FIELD](#) data structure identifies the alarm to be configured, and the `alarm_data` field specifies the configuration attribute for the alarm. (See [ALARM\\_FIELD](#), on page 367.) The set of valid values for the `alarm_data` field depends on whether the notification attribute or the blocking attribute is specified in the **alarm\_config\_type** parameter. The `alarm_data` field is of type `int`.

Parameter	Description
<b>linedev</b>	line device. The <b>linedev</b> parameter must be set to NULL to configure alarm source objects. When this value is NULL, the default behaviors for devices opened after the function call are also changed.
<b>aso_id</b>	alarm source object (ASO) ID. Use the <a href="#">gc_AlarmSourceObjectNameToID( )</a> function to obtain the ASO ID for the desired alarm source object.  The ALARM_SOURCE_ID_NETWORK_ID can be used if the network ASO ID associated with the line device is desired.
<b>alarm_list</b>	points to the alarm list. The alarm list will contain the list of alarms to be passed to the application. (See <a href="#">ALARM_LIST</a> , on page 368.) NULL is not allowed.
<b>alarm_config_type</b>	<p>performs two roles. The first role is to specify whether the application wants to be notified when any of the alarms in the <b>alarm_list</b> occur and to classify the alarms in the <b>alarm_list</b> as either blocking or non-blocking. Possible values are:</p> <ul style="list-style-type: none"> <li>ALARM_CONFIG_BLOCKING – indicates whether the specified alarms for the given <b>aso_id</b> and <b>linedev</b> are to be blocking or non-blocking. The data field for each alarm in <b>alarm_list</b> will contain either ALARM_BLOCKING (for blocking alarms) or ALARM_NONBLOCKING (for nonblocking alarms).</li> <li>ALARM_CONFIG_NOTIFY – indicates whether the application wants to be notified of the alarms in the <b>alarm_list</b> for the given <b>aso_id</b> and <b>linedev</b>. The data field for each alarm in <b>alarm_list</b> will contain either ALARM_NONNOTIFY (if the application is not to be notified) or ALARM_NOTIFY (if the application is to be notified).</li> </ul> <p>The second role is to specify whether ASOs, boards, or time slots are to be configured. For this purpose, the following values may be ORed in:</p> <ul style="list-style-type: none"> <li>ALARM_CONFIGURE_BOARDS – configure all the boards for the specified ASO ID</li> <li>ALARM_CONFIGURE_BOARDS_AND_TIMESLOTS – configure all the boards and time slots for the specified ASO ID</li> <li>ALARM_CONFIGURE_TIMESLOTS – configure the time slots for the specified board line device</li> </ul> <p>See Table 12 for more information about setting flags.</p>



Table 12. Possible Scope Settings for the `alarm_config_type` Parameter

Object(s) to Configure:			alarm_config_type Flag Setting†	linedev Setting
ASO	Board	Time Slot		
yes	yes	yes	ALARM_CONFIGURE_BOARDS_AND_TIMESLOTS	NULL
yes	yes	no	ALARM_CONFIGURE_BOARDS	NULL
yes	no	yes	configuration not supported‡	N/A
yes	no	no	none	NULL
no	yes	yes	ALARM_CONFIGURE_TIMESLOTS	Board <b>linedev</b> ID
no	yes	no	none	Board <b>linedev</b> ID
no	no	yes	none	Time slot <b>linedev</b> ID
no	no	no	configuration not supported‡	N/A
† Possible flag settings are ALARM_CONFIGURE_BOARDS_AND_TIMESLOTS, ALARM_CONFIGURE_BOARDS, ALARM_CONFIGURE_TIMESLOTS. These values must be ORed in with other values. See the description of the <b>alarm_config_type</b> parameter above. ‡ Attempts to configure unsupported configurations will return an error.				

## ■ Termination Events

None

## ■ Cautions

- When **linedev** is set to NULL, changing an alarm source object's configuration will change the default behavior for devices opened subsequent to the `gc_SetAlarmConfiguration()` function call. That is, devices opened after the `gc_SetAlarmConfiguration()` function call will get their default behavior from the alarm source object configuration information.
- Only “alarm on” attributes may be set in the **alarm\_list**. Setting the “alarm on” attribute for a specified alarm will set the “alarm off” attribute as well. For example, if DTT1\_LOS is in the list of alarms to be configured, DTT1\_LOSOK will also be configured. DTT1\_LOSOK cannot be specified in **alarm\_list**.
- If the blocking attribute of an alarm that is on is changed, the requested change will occur after the alarm clears.
- If the `gc_SetAlarmConfiguration()` function fails before it finishes updating multiple devices, that is, when **linedev** = NULL or when **linedev** = a board device, those devices that were successfully changed will remain changed (that is, the changes will not be undone).
- If a line device is specified and the given **aso\_id** is not an alarm source object for the line device, an EGC\_INVPARM error is generated.
- Although notification of a given alarm is enabled using `gc_SetAlarmConfiguration()`, the application may **not** be notified if the flow of alarms to the application has been limited by a call to `gc_SetAlarmFlow()`.

## ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*.

All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

### ■ Example

This code demonstrates setting alarm configuration for all alarms associated with a given linedevice and alarm source object

```

/*****
--      Name:      SetAlarmConfiguration()
-- Description:    According to config_type, set alarm configuration to cfgvalue
--                for all alarms on the ldev for the program-wide aso_id
--      Input:      ldev - linedevice to set
--                  if 0 then doing for aso_id
--                  aso_id - alarm source object ID
--                  config_type alarm configuration type
--                  (ALARM_CONFIG_NOTIFY or ALARM_CONFIG_BLOCKING)
--                  cfgvalue - value to set
--                  if config_type = ALARM_CONFIG_NOTIFY
--                      then valid values are ALARM_NOTIFY/ALARM_NO_NOTIFY
--                  if config_type = ALARM_CONFIG_BLOCKING
--                      then valid values are ALARM_BLOCKING/ALARM_NONBLOCKING
--      alarm_list_src_ptr - pointer to the location of the current values
--                          that are being changed. This should have been retrieved with a
--                          gc_GetAlarmConfiguration(ldev, aso_id, &alarm_status_blocking,
--                          ALARM_CONFIG_STATUS_BLOCKING)
--                          for the blocking alarms configuration value and a
--                          gc_GetAlarmConfiguration(ldev, aso_id, &alarm_status_notify,
--                          ALARM_CONFIG_STATUS_NOTIFY)
--                          for the notify configuration value
--
--      Output:      None
--      Return:       0 = success, -1 = error
*****/
int SetAlarmConfiguration(LINEDEV ldev, int aso_id, int config_type,
                          int config_value, ALARM_LIST *alarm_list_src_ptr)
{
    ALARM_LIST set_list;
    int i, rc;
    GC_INFO gc_error_info; /* GlobalCall error information data */

    switch(config_type & ~ALARM_CONFIGURE_FLAGS) /* strip "hierocracy" bits */
    {
        case ALARM_CONFIG_NOTIFY:
            /* validate config_value */
            if ((config_value != ALARM_NOTIFY) && (config_value != ALARM_NONNOTIFY))
            {
                printf("Invalid config_value (%d) in call to SetAlarmConfiguration\n",
                       config_value);
                return -1; /* ERROR RETURN POINT */
            }

            /* copy alarm_list_src_ptr to set_list and set data to desired value */
            set_list.n_alarms = alarm_list_src_ptr->n_alarms;
            for (i = 0; i < set_list.n_alarms; i++)
            {
                set_list.alarm_fields[i].alarm_number =
                    alarm_list_src_ptr->alarm_fields[i].alarm_number;
                set_list.alarm_fields[i].alarm_data.intvalue = config_value;
            }
            break;
    }
}

```

```

case ALARM_CONFIG_BLOCKING:
    /* validate config_value */
    if ((config_value != ALARM_BLOCKING) && (config_value != ALARM_NONBLOCKING))
    {
        printf("Invalid config_value (%d) in call to SetAlarmConfiguration\n",
               config_value);
        return -1; /* ERROR RETURN POINT */
    }
    /* copy alarm_list_src_ptr to set_list and set data to desired value */
    set_list.n_alarms = alarm_list_src_ptr->n_alarms;
    for (i = 0; i < set_list.n_alarms; i++)
    {
        set_list.alarm_fields[i].alarm_number =
            alarm_list_src_ptr->alarm_fields[i].alarm_number;
        set_list.alarm_fields[i].alarm_data.intvalue = config_value;
    }
    break;

default:
    printf("Invalid config_type (%d) in call to SetAlarmConfiguration\n",
           config_type);
    return -1; /* ERROR RETURN POINT */
}

/* now set alarm configuration for the desired configuration type */
rc = gc_SetAlarmConfiguration(ldev, aso_id, &set_list, config_type);

if (rc < 0)
{
    /* process error as shown */
    /* Note: gc_SetAlarmConfiguration() is a GC only function */
    /* hence the cclib error message is not printed */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_SetAlarmConfiguration() on device handle: 0x%lx,
            GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
            ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
            gc_error_info.ccLibId, gc_error_info.ccLibName,
            gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}
return(0); /* SUCCESS RETURN POINT */
}

/*
-- This code demonstrates setting the blocking values for the specific alarms
-- DTE1_LOS and DTE1_RLOS n a given linedevice that has as an alarm source object
-- ALARM_SOURCE_ID_DM3_E1
*/

LINEDEV    ldev;
int         rc;
ALARM_LIST set_list;
int         cclibid;
int         gc_error;
int         cc_error;
char        *msg;

/*
-- code assumes ldev is already initialized to the correct linedevice
-- or 0 if doing for the entire alarm source object ALARM_SOURCE_ID_DM3_E1
*/
set_list.n_alarms = 2;
set_list.alarm_fields[0].alarm_number = DTE1_LOS;

```

```
set_list.alarm_fields[0].alarm_data.intvalue = ALARM_BLOCKING;
set_list.alarm_fields[1].alarm_number = DTE1_RLOS;
set_list.alarm_fields[1].alarm_data.intvalue = ALARM_NONBLOCKING;
rc = gc_SetAlarmConfiguration(ldev, ALARM_SOURCE_ID_DM3_E1, &set_list,
                             ALARM_CONFIG_BLOCKING);

if (rc < 0)
{
    /* process error as shown */
    /* Note: gc_SetAlarmConfiguration() is a GC only function */
    /* hence the cclib error message is not printed */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_SetAlarmConfiguration() on device handle: 0x%lx,
            GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
            ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
            gc_error_info.ccLibId, gc_error_info.ccLibName,
            gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}
```

#### ■ See Also

- [gc\\_GetAlarmConfiguration\(\)](#)
- [gc\\_SetAlarmNotifyAll\(\)](#)
- [gc\\_SetAlarmFlow\(\)](#)

## `gc_SetAlarmFlow()`

**Name:** `int gc_SetAlarmFlow(aso_id, flow)`

**Inputs:** `unsigned long aso_id`      • alarm source object ID  
`int flow`      • controls the flow of alarms to the application

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** GCAMS

**Mode:** synchronous

**Platform and** Springware: T-1/E-1, ISDN

**Technology:** DM3: T-1/E-1, ISDN  
IP (host-based stack)

---

### ■ Description

The **`gc_SetAlarmFlow()`** function configures which alarms are sent to the application, that is, which alarms are allowed to “flow” to the application. This function allows the application to control the **`flow`** parameter, for example, to only allow blocking alarms, or to only allow the first alarm on and the last alarm off to be sent. This prevents the application from being flooded with alarms.

Prior to using this function, the notification attribute for alarms that the application wants to be notified of must first be turned on using the **`gc_SetAlarmConfiguration()`** or **`gc_SetAlarmNotifyAll()`** function. The **`gc_SetAlarmFlow()`** function is used to further refine which alarms are allowed to flow to the application. The default is that all candidate alarms (that is, alarms whose notify attributes are on) flow to the application.

For a list of possible alarms, see the appropriate *Global Call Technology User’s Guide*.

Parameter	Description
<code>aso_id</code>	alarm source object (ASO) ID. Use the <b><code>gc_AlarmSourceObjectNameToID()</code></b> function to obtain the ASO ID for the desired alarm source object.

Parameter	Description
<b>flow</b>	<p>flow control; for alarms whose notification attribute has been turned on, determines which of the alarms are actually sent (that is, allowed to “flow”) to the application. Possible values are:</p> <ul style="list-style-type: none"> <li>• <b>ALARM_FLOW_ALWAYS</b> – all alarms with notification on are sent to the application (default)</li> <li>• <b>ALARM_FLOW_ALWAYS_BLOCKING</b> – all blocking alarms with notification on are sent to the application. Non-blocking alarms are not sent.</li> <li>• <b>ALARM_FLOW_FIRST_AND_LAST</b> – only the first alarm on and the last alarm off are sent to the application (if their notification attribute is on). Both blocking and non-blocking alarms are eligible to be sent.</li> <li>• <b>ALARM_FLOW_FIRST_AND_LAST_BLOCKING</b> – only the first blocking alarm on and the last blocking alarm off are sent to the application (if their notification attribute is on)</li> </ul>

### ■ Termination Events

None

### ■ Cautions

The **gc\_SetAlarmFlow( )** function controls only the alarm flow, not the alarm notification. Use the **gc\_SetAlarmConfiguration( )** or **gc\_SetAlarmNotifyAll( )** function to set alarm notification on or off. The **gc\_SetAlarmFlow( )** function then further refines which alarms are allowed to flow to the application.

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

### ■ Example

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

int          rc;
GC_INFO  gc_error_info;  /* GlobalCall error information data */

/* This code assumes that the application is running over DM3 E1 */
/* and wishes to only be notified of the first and last blocking alarms */
rc = gc_SetAlarmFlow(ALARM_SOURCE_ID_DM3_E1, ALARM_FLOW_FIRST_AND_LAST_BLOCKING);
if (rc < 0)
{
    /* process error as shown */
}
```

```
gc_ErrorInfo( &gc_error_info );
printf ( "Error: gc_SetAlarmFlow(), GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s,
        CC ErrorValue: 0x%lx - %s\n",
        gc_error_info.gcValue, gc_error_info.gcMsg,
        gc_error_info.ccLibId, gc_error_info.ccLibName,
        gc_error_info.ccValue, gc_error_info.ccMsg);
return (gc_error_info.gcValue);
}
```

#### ■ See Also

- [gc\\_GetAlarmFlow\(\)](#)
- [gc\\_SetAlarmConfiguration\(\)](#)
- [gc\\_SetAlarmNotifyAll\(\)](#)

## gc\_SetAlarmNotifyAll( )

**Name:** int gc\_SetAlarmNotifyAll(linedev, aso\_id, value)

**Inputs:** LINEDEV linedev      • line device  
          unsigned long aso\_id      • alarm source object ID  
          int value      • notification value to set

**Returns:** 0 if successful  
          <0 if failure

**Includes:** gclib.h  
          gcerr.h

**Category:** GCAMS

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN

DM3: T-1/E-1, ISDN  
IP (host-based stack)

---

### ■ Description

The **gc\_SetAlarmNotifyAll()** function sets the notification attribute of all alarms originating from a specified alarm source object.

The **gc\_SetAlarmNotifyAll()** function can be used as a shortcut for changing the notify attribute, instead of using the **gc\_GetAlarmConfiguration()** function to retrieve the list of all alarms and then setting their notify attribute using the **gc\_SetAlarmConfiguration()** function.

The **gc\_SetAlarmNotifyAll()** function can also be used to determine if an alarm exists on a line device when the device is opened. After calling **gc\_OpenEx()**, enable alarm notification for the line device. If any alarms are already active on the line device, the application will be notified with one or more applicable GCEV\_ALARM events, depending on the current value of the alarm flow control.

Parameter	Description
<b>linedev</b>	line device. The <b>linedev</b> parameter must be set to NULL to set the notification attribute for alarm source objects. Setting <b>linedev</b> to NULL changes the default behavior for any device opened after the function call.



Parameter	Description
<b>aso_id</b>	alarm source object (ASO) ID. Use the <code>gc_AlarmSourceObjectNameToID()</code> function to obtain the ASO ID for the desired alarm source object.  ALARM_SOURCE_ID_NETWORK_ID can be used if the network ASO ID is not known.
<b>value</b>	<p>performs two roles. The first role is to set the alarm notification attribute. Possible values are:</p> <ul style="list-style-type: none"> <li>ALARM_NOTIFY – notify the application of all alarms in <b>alarm_list</b></li> <li>ALARM_NONOTIFY – do not notify the application of alarms in <b>alarm_list</b></li> </ul> <p>The second role is to define whether the notification attribute is to be set for all boards or time slots associated with the ASO or for a specific board associated with the ASO, allowing users to control the extent of the changes. The following flags, which may be ORed in, can be used to define whether which ASO IDs, boards, or time slots are to be configured:</p> <ul style="list-style-type: none"> <li>ALARM_CONFIGURE_BOARDS – configure all the boards for the specified ASO ID</li> <li>ALARM_CONFIGURE_BOARDS_AND_TIMESLOTS – configure all the boards and time slots for the specified ASO ID</li> <li>ALARM_CONFIGURE_TIMESLOTS – configure the time slots for the specified board</li> </ul> <p>See Table 13 for more information about setting these flags. Only one of the three flags may be set at a time.</p>

**Table 13. Possible Flag Settings for value Parameter**

Object(s) to Configure:			alarm_config_type Flag Setting†	linedev Setting
ASO	Board	Time Slot		
yes	yes	yes	ALARM_CONFIGURE_BOARDS_AND_TIMESLOTS	NULL
yes	yes	no	ALARM_CONFIGURE_BOARDS	NULL
yes	no	yes	configuration not supported‡	N/A
yes	no	no	none	NULL
no	yes	yes	ALARM_CONFIGURE_TIMESLOTS	Board <b>linedev</b> ID
no	yes	no	none	Board <b>linedev</b> ID
no	no	yes	none	Time slot <b>linedev</b> ID
no	no	no	configuration not supported‡	N/A
† Possible flag settings are ALARM_CONFIGURE_BOARDS_AND_TIMESLOTS, ALARM_CONFIGURE_BOARDS, ALARM_CONFIGURE_TIMESLOTS. ‡ Attempts to configure unsupported configurations will return an error.				

## ■ Termination Events

None

## ■ Cautions

- When **linedev** is set to NULL, changing an alarm source object's configuration will change the default behavior for devices opened subsequent to the **gc\_SetAlarmNotifyAll()** function call. That is, devices opened after the **gc\_SetAlarmNotifyAll()** function call will get their default behavior from the alarm source object configuration information.
- If a line device is specified and the given **aso\_id** is not an alarm source object for the line device, an error will occur.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdner.h* file for the ISDN call control library).

## ■ Example

```
/* Example 1: change ASO, but not any open devices */
int      rc;
/* Enable alarm notification on all Springware E1 alarms */
/* Note: this does not change any open devices */
rc = gc_SetAlarmNotifyAll(0, ALARM_SOURCE_ID_SPRINGWARE_E1, ALARM_NOTIFY);
if (rc < 0)
{
    /* get and process the error */
}

/* Example 2: change ASO, and all open devices that use this ASO */
int      rc;
/* Enable alarm notification on all Springware E1 alarms */
/* Note: this also changes all open devices associated that use this ASO */
rc = gc_SetAlarmNotifyAll(0, ALARM_SOURCE_ID_SPRINGWARE_E1, ALARM_NOTIFY |
                          ALARM_CONFIGURE_BOARDS_AND_TIMESLOTS);
if (rc < 0)
{
    /* get and process the error */
}
```

## ■ See Also

- [gc\\_GetAlarmConfiguration\(\)](#)
- [gc\\_SetAlarmConfiguration\(\)](#)

## `gc_SetAlarmParm()`

**Name:** `int gc_SetAlarmParm(linedev, aso_id, ParmSetID, alarm_parm_list, mode)`

**Inputs:**

<code>LINEDEV linedev</code>	• line device
<code>unsigned long aso_id</code>	• alarm source object ID
<code>int ParmSetID</code>	• parameter set ID
<code>ALARM_PARM_LIST *alarm_parm_list</code>	• pointer to alarm parameter list
<code>unsigned long mode</code>	• sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** GCAMS

**Mode:** synchronous

**Platform and** Springware: T-1/E-1, ISDN

**Technology:** IP (host-based stack† and embedded stack†)

†See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

The `gc_SetAlarmParm()` function sets the data associated with the alarm parameter for one or more alarm number and parameter ID pairs. The alarm number and parameter set ID pairs are specified in `ALARM_PARM_FIELD` in the **`alarm_parm_list`** list.

The parameter set ID identifies a set of associated parameters as defined by the alarm source object (ASO). An example of one parameter set is the parameters that can be set by the `dt_setparm()` function. Another example of a parameter set is the parameters that are set by the `dt_setalarm()` function. Both functions are in the Springware DT library.

Examples of alarm parameters that can be set include timing parameters and maximum number of errors. Parameters can be set on either a line device basis or an ASO ID basis.

Parameter	Description
<b><code>linedev</code></b>	Global Call line device handle
<b><code>aso_id</code></b>	alarm source object (ASO) ID. Use the <a href="#">gc_AlarmSourceObjectNameToID()</a> function to obtain the ASO ID for the desired alarm source object.  <code>ALARM_SOURCE_ID_NETWORK_ID</code> can be used if the network ASO ID is not known.
<b><code>ParmSetID</code></b>	identifies a set of associated parameters

Parameter	Description
<b>alarm_parm_list</b>	points to the alarm parameter list. See <a href="#">ALARM_PARM_LIST</a> , on page 370 for a description of the fields in the ALARM_PARM_LIST data structure.
<b>mode</b>	set to EV_SYNC for synchronous mode (only synchronous mode is supported)

## ■ Termination Events

None

## ■ Cautions

- The alarm parameters are alarm source object dependent. Detailed knowledge of the alarm source object is necessary in order to use these parameters properly.
- The exact usage of the ALARM\_PARM\_FIELD of **alarm\_parm\_list** is dependent upon (alarm source object, parameter set ID) pair.
- It is important that the Global Call API is able to call the alarm source object's underlying functions with the exact same syntax used by the functions. To ensure proper use of the alarm source object variables, see the appropriate documentation.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdner.h* file for the ISDN call control library).

## ■ Example

```
#include <stdio.h>
#include <stdlib.h>
#include <gclib.h>
#include <gcerr.h>
#include <dtilib.h>          /* for ASO symbols */

int main(int argc, char **argv)
{
    int          rc;
    ALARM_PARM_LIST alarm_parm_list;

    /* perform application initialization */

    /* assumes gc_Start() has already been successfully called */

    /* init all to 0 */
    memset(&alarm_parm_list, '\0', sizeof(ALARM_PARM_LIST));

    /* set 1 parameter */
    alarm_parm_list.n_parms = 1;

    /* set # of out of frame errors to allow before sending an alarm */
    alarm_parm_list.alarm_parm_fields[0].alarm_parm_number.intvalue = DTG_OOFMAX;
```

```

/* 3 is only used as an example and has no significance */
alarm_parm_list.alarm_parm_fields[0].alarm_parm_data.intvalue = 3;

/* Set for all of Springware T1 */
rc = gc_SetAlarmParm(0, ALARM_SOURCE_ID_SPRINGWARE_T1, ParmSetID_parm,
                    &alarm_parm_list, EV_SYNC);

if (rc < 0)
{
    /* get and process the error */
}

/* continue with the application */
}

```

#### ■ See Also

- [gc\\_GetAlarmParm\(\)](#)

## gc\_SetBilling( )

**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

**Category:** optional call handling

**Mode:** asynchronous or synchronous

**Platform and Technology:** Springware: T-1/E-1†, ISDN†

SS7†

†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The **gc\_SetBilling( )** function sets billing information associated with the specified call reference number (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-A-Bill service. For some E-1 CAS protocols, different billing rates can be chosen on a per-call basis.

See also the appropriate *Global Call Technology User's Guide* for technology-specific information about the **rate\_type**, **ratep**, and **mode** parameters.

Parameter	Description
<b>crn</b>	call reference number
<b>rate_type</b>	type of billing data
<b>ratep</b>	points to the GC_RATE_U data structure, which contains the charge information for the current call. For a description of the structure, see <a href="#">GC_RATE_U</a> , on page 395.
<b>mode</b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution (see the appropriate <i>Global Call Technology User's Guide</i> to determine if the asynchronous mode is supported)

### ■ Termination Events

GCEV\_SETBILLING

indicates that the function was successful, that is, the charge information was set.

## GCEV\_TASKFAIL

indicates that the function failed. For more information, see the “Error Handling” section in the *Global Call API Programming Guide*.

### ■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value `EGC_UNSUPPORTED` will be the Global Call value returned when the `gc_ErrorInfo()` function is used to retrieve the error code.

### ■ Errors

If this function returns `<0` to indicate failure, use the `gc_ErrorInfo()` function for error information. If the `GCEV_TASKFAIL` event is received, use the `gc_ResultInfo()` function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

### ■ 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) has been
 *    called to convert the event into metaevent.
 * 4. a GCEV_OFFERED has been detected.
 * 5. a call has been established.
 */

#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 any time after the connection
 *       is established.
 */
int set_billing(CRN crn, int rate_type, GC_RATE_U *ratep, unsigned long mode)
{
    LINEDEV ldev;          /* Line device */
    GC_INFO gc_error_info; /* GlobalCall error information data */

    if(gc_CRN2LineDev(crn, &ldev) != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_CRN2LineDev() on crn: 0x%x, GC ErrorValue: 0x%x - %s,\n",
                CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
```

```
        crn, gc_error_info.gcValue, gc_error_info.gcMsg,  
        gc_error_info.ccLibId, gc_error_info.ccLibName,  
        gc_error_info.ccValue, gc_error_info.ccMsg);  
    return (gc_error_info.gcValue);  
}  
  
if(gc_SetBilling(crn, rate_type, ratep, mode) != GC_SUCCESS) {  
    gc_ErrorInfo( &gc_error_info );  
    printf ("Error: gc_SetBilling() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,  
           CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",  
           ldev, gc_error_info.gcValue, gc_error_info.gcMsg,  
           gc_error_info.ccLibId, gc_error_info.ccLibName,  
           gc_error_info.ccValue, gc_error_info.ccMsg);  
    return (gc_error_info.gcValue);  
}  
return(0);  
}
```

**■ See Also**

- [gc\\_SetParm\(\)](#)



## `gc_SetCallingNum()`

**Name:** `int gc_SetCallingNum(linedev, calling_num)`

**Inputs:** `LINEDEV linedev` • Global Call line device handle  
`char *calling_num` • pointer to calling phone number string

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gcLib.h`  
`gcerr.h`

**Category:** optional call handling

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN†

**Technology:** DM3: T-1/E-1, ISDN†, Analog  
SS7  
IP (embedded stack)

†See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

**Note:** For technologies that support the `gc_SetConfigData()` function, it is recommended that the `gc_SetConfigData()` function be used instead of the `gc_SetCallingNum()` function to set calling numbers. See the `gc_SetConfigData()` function description for more information.

The `gc_SetCallingNum()` function sets the 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
<code>linedev</code>	Global Call line device handle
<code>calling_num</code>	phone number of the calling party (ASCII string format)

### ■ Termination Events

None

### ■ Cautions

None

### ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology

specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

### ■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <gcLib.h>
#include <gcerr.h>

int set_calling_num(LINEDEV ldev)
{
    GC_INFO      gc_error_info;    /* GlobalCall error information data */

    /* Set up the calling party number on the line device */
    if (gc_SetCallingNum(ldev, "2019933000") != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_SetCallingNum() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    /*
     * 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);
}
```

### ■ See Also

- [gc\\_MakeCall\(\)](#)
- [gc\\_SetConfigData\(\)](#)

## `gc_SetCallProgressParm()`

**Name:** `int gc_SetCallProgressParm(linedev, parm)`

**Inputs:** `LINEDEV linedev` • Global Call line device handle  
`DX_CAP *parm` • pointer to the call analysis information structure

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** system controls and tools

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1 (PDKRT only)

---

### ■ Description

The `gc_SetCallProgressParm()` function allows the application to specify the values of the configurable call progress parameters, which override default call progress parameters provided by the protocol. The parameters are specified in the `DX_CAP` data structure.

Parameter	Description
<code>linedev</code>	Global Call line device handle
<code>parm</code>	pointer to the call progress information structure, <code>DX_CAP</code> . For a description of the structure, see <code>DX_CAP</code> , on page 375.

### ■ Termination Events

None

### ■ Cautions

None

### ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <gcLib.h>
#include <gcerr.h>

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

/*
 * Data structure which stores all information for each line
 */
static struct linebag {
    LINEDEV  ldev;          /* GlobalCall API line device handle */
    CRN      crn;           /* GlobalCall API call handle */
    int      blocked;       /* channel blocked/unblocked */
    int      networkh;      /* network handle */
} port[MAX_CHAN+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 and voice handle is stored in linebag structure "port"
 */
int call_setcallprogressparm(int port_num, DX_CAP *infop)
{
    GC_INFO  gc_error_info;  /* GlobalCall error information data */

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

    /* Set the number of maximum rings */
    infop->ca_nbrdna = 5;

    /* Set information about the device and store it in the location pointed
       to by infop */
    if (gc_SetCallProgressParm(pline->ldev, infop) == -1)
    {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_SetCallProgressParm() on device handle: 0x%x,
                GC ErrorValue: 0x%x - %s, CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
                pline->ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    return (0);
}
```

## ■ See Also

- [gc\\_GetCallProgressParm\( \)](#)

## `gc_SetChanState()`

**Name:** `int gc_SetChanState(linedev, chanstate, mode)`

**Inputs:**

<code>LINEDEV linedev</code>	• Global Call line device handle
<code>int chanstate</code>	• channel service state
<code>unsigned long mode</code>	• async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** optional call handling

**Mode:** asynchronous or synchronous

**Platform and Technology:** Springware: T-1/E-1†, ISDN PRI (T-1 only)†

DM3: T-1/E-1†, ISDN†, Analog  
SS7†

†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The `gc_SetChanState()` function sets the channel state of the indicated channel. When power is initially applied, all channels are placed in the In Service state.

See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>chanstate</b>	service state of line. Possible values are: <ul style="list-style-type: none"> <li>• <code>GCLS_INSERVICE</code> – inform driver that host is ready to receive and send a message</li> <li>• <code>GCLS_MAINTENANCE</code> – inform host that normal outbound traffic is not allowed and that only inbound test calls can be made</li> <li>• <code>GCLS_OUT_OF_SERVICE</code> – inform driver that host is not ready to receive or send messages. See the appropriate <i>Global Call Technology User's Guide</i> for inbound and outbound requests that will be rejected.</li> </ul>
<b>mode</b>	set to <code>EV_ASYNC</code> for asynchronous execution or to <code>EV_SYNC</code> for synchronous execution

### ■ Termination Events

`GCEV_SETCHANSTATE`

indicates that the function was successful, that is, the request for a change in channel state was accepted.

#### **GCEV\_TASKFAIL**

indicates that the function failed. For more information, see the “Error Handling” section in the *Global Call API Programming Guide*.

### ■ Cautions

- This function should only be invoked while in the Null call state. The Null call state occurs when a call to the **gc\_OpenEx( )**, **gc\_ReleaseCallEx( )** or **gc\_ResetLineDev( )** function is terminated.
- Do not call **gc\_WaitCall( )** or **gc\_MakeCall( )** until this function completes.
- It is the responsibility of the application to recognize that the **gc\_SetChanState( )** function was used to put a channel in the maintenance (GCLS\_MAINTENANCE) or the out-of-service (GCLS\_OUT\_OF\_SERVICE) state. A GCEV\_BLOCKED event is not necessarily generated. Do not call **gc\_WaitCall( )** or **gc\_MakeCall( )** while a channel is in the maintenance or out-of-service state.

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function for error information. If the GCEV\_TASKFAIL event is received, use the **gc\_ResultInfo( )** function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

### ■ Example

```
#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_ReleaseCallEx() function.
 */

int set_channel_InService(LINEDEV ldev)
{
    int          state;          /* State to which channel has to be set */
    GC_INFO  gc_error_info;      /* GlobalCall error information data */

    /*
     * 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_ErrorInfo( &gc_error_info );
        printf ("Error: gc_SetChanState() on device handle: 0x%x,
                GC ErrorValue: 0x%x - %s, CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
                ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
}
```

```
/*  
 * Application can change state again when necessary.  
 */  
return (0);  
}
```

#### ■ See Also

- [gc\\_WaitCall\(\)](#)

## gc\_SetConfigData( )

**Name:** int gc\_SetConfigData(target\_type, target\_id, target\_datap, time\_out, update\_cond, request\_idp, mode)

<b>Inputs:</b> int target_type	• target object type
long target_id	• target object ID
GC_PARM_BLK target_datap	• pointer to the location of the configuration data used to update the target object data
int time_out	• time-out in seconds.
int update_cond	• when update will happen
long * request_idp	• pointer to the location for storing request ID
unsigned long mode	• async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** gcerr.h

**Category:** RTCM, system controls and tools

**Mode:** asynchronous or synchronous

**Platform and Technology:** Springware: T-1/E-1 (PDKRT only)†, ISDN†, Analog (PDKRT only)

SS7†

IP (host-based stack† and embedded stack†)

†See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

The **gc\_SetConfigData( )** function supports the Global Call Real Time Configuration Management (RCM) feature. The **gc\_SetConfigData( )** updates the configuration data for a given target object.

Configuration data for multiple parameters can be updated in a single function call. Only one target object can be accessed in a single function call.

See [Table 22, “GC Parameter Entry List Maintained in GCLIB”](#), on page 422 and [Table 23, “Examples of Parameter Entry List Maintained in CCLIB”](#), on page 424. For more information about the RTCM feature, see the *Global Call API Programming Guide*.

**Note:** For those technologies that support the **gc\_SetConfigData( )** function, **gc\_SetConfigData( )** is the preferred alternative to the **gc\_SetCallingNum( )** and **gc\_SetEvtMsk( )** functions. For parameters used to set event masks, see [Table 22, “GC Parameter Entry List Maintained in GCLIB”](#), on page 422. For examples of parameters used to set calling numbers, see [Table 23, “Examples of Parameter Entry List Maintained in CCLIB”](#), on page 424.



Parameter	Description
<b>target_type</b>	target object type
<b>target_id</b>	target object identifier. This identifier, along with <b>target_type</b> , uniquely specifies the target object.
<b>target_datap</b>	points to the GC_PARM_BLK structure. This structure contains the parameter configuration data to be updated. See <a href="#">GC_PARM_BLK</a> , on page 392 for more information.
<b>time_out</b>	time interval (in seconds) during which the target object must be updated with the data; if the interval is exceeded, the request to update is ignored. This parameter is ignored when set to 0.  This parameter is supported in synchronous mode only. If the update request is not completed in the time specified, the <code>gc_SetConfigData()</code> function will return EGC_TIMEOUT.
<b>update_cond</b>	specifies when to update. Valid values are: <ul style="list-style-type: none"> <li>GCUPDATE_IMMEDIATE – requires parameter to be updated immediately</li> <li>GCUPDATE_ATNULL – requires parameter to be updated at the Null call state. Applicable only if the <b>target_type</b> is associated with a call.</li> </ul> <p><b>Note:</b> The GCUPDATE_ATNULL value should be used only if the <code>gc_SetConfigData()</code> function is called in asynchronous mode, that is, with the <b>mode</b> parameter set to EV_ASYNC.</p>
<b>request_idp</b>	points to the location for storing the request ID, which is generated by Global Call
<b>mode</b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

The parameter **target\_datap** points to the location of the [GC\\_PARM\\_BLK](#), which stores the configuration parameter data to be updated. Memory allocation and deallocation of the GC\_PARM\_BLK data block is done by the Global Call utility functions (`gc_util_XXX`). See [Section 1.12, “GC\\_PARM\\_BLK Utility Functions”](#), on page 18 for more information.

It is the Global Call application’s responsibility to use the Global Call utility functions to allocate an appropriate-size data block memory ([GC\\_PARM\\_BLK](#)) for the configuration parameters and to insert parameter information (such as the set ID, parm ID, value buffer size, value buffer, and value data) into the GC\_PARM\_BLK data block. After successfully calling the `gc_SetConfigData()` function, the parameter value(s) in the GCLib or CCLib are updated with the values given in the GC\_PARM\_BLK (value buffer fields). After finishing its use of the GC\_PARM\_BLK, the Global Call application should deallocate the GC\_PARM\_BLK data block using the [gc\\_util\\_delete\\_parm\\_blk\(\)](#) function. See [GC\\_PARM\\_BLK](#), on page 392 for more information.

The function outputs a unique request ID to verify received update events and query the update results. All subsequent references to this request must be made using the request ID.

The parameter configuration data can be either read-only, update immediately, or update at the Null call state. When the `gc_SetConfigData()` function is issued, the action taken by the application depends on the update condition of the parameter, as described in Table 14.

Table 14. Parameter Configuration Data Conditions and Results

Parameter Update Condition	Result when gc_SetConfigData( ) is Called
Read-only	An error is returned immediately.
Update immediately	Configuration data is updated immediately.
Update only at Null call state	Configuration data is updated when the target object does not have an active call (Null call state). A request for an immediate update may fail. To immediately update parameters that can be updated only at the Null call state, call <b>gc_SetConfigData( )</b> with GCUPDATE_ATNULL and then call <b>gc_ResetLineDev( )</b> .

The **gc\_SetConfigData( )** function supports both the synchronous and asynchronous modes. In the asynchronous mode, the termination events, GCEV\_SETCONFIGDATA and GCEV\_SETCONFIGDATA\_FAIL, both have an associated **GC\_RTCM\_EVTDATA** data structure (to which the evtdatap field in **METAEVENT** points) that includes the request ID, additional message, etc., but without the retrieved **GC\_PARM\_BLK** field. The request ID is used by the application to trace the function call. See **GC\_RTCM\_EVTDATA**, on page 396 for more information.

When the **gc\_SetConfigData( )** function is called to update the configuration data of a group of parameters, the request will terminate on any single parameter updating failure. Some parameters may be updated while others may not be updated. To find out what kind of error occurred and which parameter data pair (set ID, parm ID) were being updated when it occurred, use the **gc\_ErrorInfo( )** function to retrieve information about function return values or the **gc\_ResultInfo( )** function to retrieve information about termination events. (See the “Error Handling” section in the *Global Call API Programming Guide* for more information about these error functions.) To determine which parameters were updated, use the **gc\_GetConfigData( )** function to retrieve the current configuration. After the error is fixed, call the **gc\_SetConfigData( )** function again to update the remaining parameters.

### ■ Termination Events

GCEV\_SETCONFIGDATA

indicates that the **gc\_SetConfigData( )** function was successful, that is, the configuration data has been successfully updated

GCEV\_SETCONFIGDATA\_FAIL

indicates that the **gc\_SetConfigData( )** function failed

### ■ Cautions

- Parameters should be set with caution. Improper setting of parameters may result in unpredictable behavior. Consult the *Global Call API Programming Guide* and the appropriate *Global Call Technology User's Guide* for the impact of changing any settings.
- In the synchronous mode, the function does not return until the request is completed. If the parameter is allowed to update only at the Null call state (no active call) but the current call state for the related target object is not at the Null call state, the system may be blocked. This occurs because the function waits for the Null call state while the system waits for the function to finish. To avoid this blocking, use the asynchronous mode.

- Only synchronous mode is supported for the following target objects:  
GCTGT\_GCLIB\_SYSTEM, GCTGT\_CCLIB\_SYSTEM, GCTGT\_PROTOCOL\_SYSTEM,  
and GCTGT\_FIRMWARE\_NETIF. Otherwise, the function will return the async mode error.

## ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function for error information. If the GCEV\_SETCONFIGDATA\_FAIL event is received, use the `gc_ResultInfo()` function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

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

#define MAXCHAN 30
/*
Data structure that stores all information about each line device
*/
struct linebag
{
    LINEDEV    ldev;          /* Lline device ID */
    CRN        crn;           /* CRN */
    int        call_sate;
} port[MAXCHAN];

struct linebag    *pline;

/* set the configuration of a GC target object */
int set_configuration(int target_type, int port_num, GC_PARM_BLK * parm_blkp,
                    int update_flag, long * request_idp, unsigned long mode)
{
    int    result;
    long   target_id = 0;
    GC_INFO t_gcinfo;

    switch (target_type)
    {
        case GCTGT_GCLIB_SYSTEM:
            target_id = GC_LIB;
            break;
        case GCTGT_CCLIB_SYSTEM:
            /*get cclib ID */
            break;
        case GCTGT_PROTOCOL_SYSTEM:
            /*get protocol ID by calling gc_QueryConfigData() with protocol name */
            break;
        case GCTGT_GCLIB_CRN:
        case GCTGT_CCLIB_CRN:
            /* If the target object is a CRN */
            pline = port + port_num;
            target_id = pline->crn;
            break;
        case GCTGT_GCLIB_NETIF:
        case GCTGT_CCLIB_NETIF:
        case GCTGT_GCLIB_CHAN:
        case GCTGT_CCLIB_CHAN:
```

```

        case GCTGT_PROTOCOL_CHAN:
        case GCTGT_FIRMWARE_CHAN:
            /* If the target object is a line device (time slot or network interface) */
            pline = port + port_num;
            target_id = pline->ldev;
            break;
        default:
            /* Otherwise: return -1 */
            printf("Unsupported target type: 0x%lx\n", target_type);
            return -1;
            break;
    }

    /* Call gc_SetConfigData() function */
    result = gc_SetConfigData(target_type, target_id, parm_blkp, 0, update_flag,
                             request_idp, mode);
    if (result != GC_SUCCESS)
    {
        /* retrieve error values by calling gc_ErrorInfo */
        if (gc_ErrorInfo(&t_gcinfo) == GC_SUCCESS)
        {
            printf("Error on target type: 0x%lx, target ID: 0x%lx\n", target_type,
                   target_id);
            printf("with GC Error 0x%xh: %s\n", t_gcinfo.gcValue, t_gcinfo.gcMsg);
            printf("CCLib %d(%s) Error - 0x%xh: %s\n", t_gcinfo.ccLibId,
                   t_gcinfo.ccLibName, t_gcinfo.ccValue, t_gcinfo.ccMsg);
            printf("Additional message: %s\n", t_gcinfo.additionalInfo);
        }
        else
        {
            printf("gc_ErrorInfo() failure");
        }
    }
    return result;
}

int main()
{
    int    port_num = 0;
    long   request_id = 0;
    long   long_value;

    /* To call the GC PARM utility function to insert a parameter,
       the pointer to GC_PARM_BLK must be initialized to NULL */
    GC_PARM_BLK * parm_data_blkp = NULL;

    /* First call GC PARM utility function to insert the parameters used to be
       updated */

    /* 1. insert set call event mask parm */
    long_value = GCMSK_DETECTED | GCMSK_DIALING | GCMSK_FATALERROR | GCMSK_UNBLOCKED;
    gc_util_insert_parm_val(&parm_data_blkp, GCSET_CALLEVENT_MSK, GCACT_SETMSK,
                           sizeof(long), long_value);

    /* 2. insert set call state mask parm */
    long_value = GCMSK_PROCEEDING_STATE | GCMSK_SENDMOREINFO_STATE |
                 GCMSK_GETMOREINFO_STATE | GCMSK_CALLROUTING_STATE;
    gc_util_insert_parm_val(&parm_data_blkp, GCSET_CALLSTATE_MSK, GCACT_SETMSK,
                           sizeof(long), long_value);

    /* after creating the GC_PARM_BLK, call set_configuration function to set the target
       object with the data block */
    set_configuration(GCTGT_GCLIB_CHAN, port_num, parm_data_blkp, GCUPDATE_IMMEDIATE,
                     &request_id, EV_SYNC);
}

```

```
/* delete the parm data block after using it */  
gc_util_delete_parm_blk(parm_data_blkp);  
return 0;  
}
```

#### ■ See Also

- [gc\\_GetConfigData\(\)](#)
- [gc\\_QueryConfigData\(\)](#)

## gc\_SetEvtMsk( )

**Name:** int gc\_SetEvtMsk(linedev, mask, action)

**Inputs:**

LINEDEV linedev	• Global Call line device handle
unsigned long mask	• 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 controls and tools

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1†, ISDN†, Analog

DM3: T-1/E-1, ISDN†, Analog  
SS7  
IP (embedded stack)

†See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

**Note:** For technologies that support the **gc\_SetConfigData( )** function, it is recommended that the **gc\_SetConfigData( )** function be used instead of the **gc\_SetEvtMsk( )** function to set event masks. See the **gc\_SetConfigData( )** function description for more information.

The **gc\_SetEvtMsk( )** function sets the event mask associated with a specified line device. If an event **mask** 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, for example, a time slot. See the appropriate *Global Call Technology User's Guide* to determine the level of the event masks needed.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>mask</b>	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 <b>mask</b> values are listed in Table 15.

Parameter	Description
<b>action</b>	<p>application may either set or reset the mask bit(s) as specified in <b>mask</b>. Possible actions are:</p> <ul style="list-style-type: none"> <li>• <b>GCACT_SETMSK</b> – enables notification of events specified in bitmask parameter and disables notification of any event not specified.</li> <li>• <b>GCACT_ADDMSK</b> – adds notification of events specified in bitmask parameter to previously enabled events.</li> <li>• <b>GCACT_SUBMSK</b> – disables notification of events specified in bitmask parameter.</li> </ul>

**Table 15. mask Parameter Values**

Type	Description	Default
GCMASK_ALERTING	Set mask for the GCEV_ALERTING event.	Enabled
GCMASK_BLOCKED	Set mask for the GCEV_BLOCKED event.	Enabled
GCMASK_DETECTED	Set mask for the GCEV_DETECTED event.	Disabled
GCMASK_DIALING	Set mask for the GCEV_DIALING event.	Disabled
GCMASK_DIALTONE	Set mask for the GCEV_DIALTONE event.	Disabled
GCMASK_REQMOREINFO	Set mask for the GCEV_REQMOREINFO event.	Disabled
GCMASK_PROCEEDING	Set mask for the GCEV_PROCEEDING event.	Disabled
GCMASK_UNBLOCKED	Set mask for the GCEV_UNBLOCKED event.	Enabled
<b>Note:</b> Not all events are supported by each technology, for example, the GCEV_DIALING event is supported in applications that use PDKRT protocols but is not supported in applications that use ICAPI protocols.		

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) the 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 appropriate *Global Call Technology User's Guide* for additional details.

## ■ Termination Events

None

## ■ Cautions

In the event of a failure, it is possible that there may be partial success; that is, some masks may be set or reset while others are not.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnnerr.h* file for the ISDN call control library).

## ■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <gcclib.h>
#include <gcerr.h>

/*
 * Assume the following has been done:
 * 1. The line device has been opened.
 */
int set_event_mask(LINEDEV ldev)
{
    GC_INFO gc_error_info; /* GlobalCall error information data */

    /*
     * Set the event Detected and Dialing event masks to enable
     * for this application.
     */
    /*
     * Enable the Detected and Dialing events.
     */
    if (gc_SetEvtMsk(ldev, (GCMASK_DETECTED | GCMASK_DIALING), GCACT_ADDMSK)
        != GC_SUCCESS) {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_SetEvtMsk() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                ldev, gc_error_info.gcValue, gc_error_info.gCMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    /*
     * Proceed to generate or accept calls on this line device.
     */

    /*
     * Now disable notification of Detected and Dialing events,
     * and enable notification of Alerting event, without
     * affecting any other event masks which may have been set.
     */
}
```





*set the event mask associated with a specified line device — `gc_SetEvtMsk()`*

```
if (gc_SetEvtMsk(ldev, (GCMSK_DETECTED | GCMSK_DIALING), GCACT_SUBMSK)
    != GC_SUCCESS) {
    /* Process error */
}
if (gc_SetEvtMsk(ldev, GCMSK_ALERTING, GCACT_ADDMSK) != GC_SUCCESS) {
    /* Process error */
}
return (0);
}
```

#### ■ See Also

- [gc\\_SetConfigData\(\)](#)
- [gc\\_SetParm\(\)](#)

## gc\_SetInfoElem( )

**Name:** int gc\_SetInfoElem(linedev, iep)

**Inputs:** LINEDEV linedev • B channel Global Call 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

**Category:** interface specific

**Mode:** synchronous

**Platform and Technology:** Springware: ISDN†

SS7†  
†See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

**Note:** The **gc\_SetInfoElem( )** function is deprecated in this software release. The suggested equivalent is **gc\_SetUserInfo( )**.

The **gc\_SetInfoElem( )** function allows applications to set an additional information element in the next outbound ISDN message on a specific B 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.

See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>iep</b>	points to the starting address of the information element data structure; see <a href="#">GC_IE_BLK</a> , on page 387

### ■ Termination Events

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 Global Call value returned when the [gc\\_ErrorInfo\(\)](#) function is used to retrieve the error code.

## ■ Errors

If this function returns `<0` to indicate failure, use the [gc\\_ErrorInfo\(\)](#) function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
/*
 * Assume the following has been done:
 * 1. device has been opened (e.g. :N_dtiB1T1:P_isdn, :N_dtiB1T2:P_isdn, etc...)
 */
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>

GC_IE_BLK    gc_ie_blk;
GCLIB_IE_BLK gclib_ie_blk;
IE_BLK       ie_blk;
GC_INFO      gc_error_info; /* GlobalCall error information data */

gc_ie_blk.gclib = &gclib_ie_blk;
gc_ie_blk.cclib = &ie_blk;

ie_blk.length = 6;
ie_blk.data[0] = 0x7E;
ie_blk.data[1] = 0x4;
ie_blk.data[2] = 0x01;
ie_blk.data[3] = 0x01;
ie_blk.data[4] = 0x02;
ie_blk.data[5] = 0x04;

if (gc_SetInfoElem (ldev, &gc_ie_blk) < 0)
{
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_SetInfoElem() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
           CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
           ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
           gc_error_info.ccLibId, gc_error_info.ccLibName,
           gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}
```

## ■ See Also

- [gc\\_SetParm\(\)](#)
- [gc\\_SetUserInfo\(\)](#)

## gc\_SetParm( )

**Name:** int gc\_SetParm(linedev, parm\_id, value)

**Inputs:**

LINEDEV linedev	• Global Call 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

**Category:** systems control and tools

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1†, ISDN†, Analog† (not supported when using PDK analog)  
DM3: T-1/E-1†, ISDN†, Analog†  
SS7†  
†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The **gc\_SetParm( )** function sets the default parameters and all channel information associated with the specific line device.

See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>parm_id</b>	parameter ID of the parameter to be set. The parameter ID definitions for parameters that are common across all technologies are listed in Table 16. The “Level” column indicates whether the parameter is a channel level parameter or a trunk level parameter. To set a trunk level parameter, the <b>linedev</b> parameter must be the device ID associated with a network interface trunk. See <b>GC_PARM</b> , on page 391 for data structure details. See the appropriate <i>Global Call Technology User's Guide</i> for additional parameter IDs.
<b>value</b>	value selected for parameter being set

**Table 16. Parameter Descriptions, `gc_GetParm()` and `gc_SetParm()`**

Parameter†	Level	Description
GCPR_CALLINGPARTY	channel	Calling party number (pointer to null-terminated ASCII string) (possible values are the existing GTD identification numbers). Use address field of <code>GC_PARM</code> .
† See the appropriate <i>Global Call Technology User's Guide</i> for additional parameters.		

## ■ Termination Events

None

## ■ Cautions

The `gc_SetParm()` function can be called only once in the application to set the `RECEIVE_INFO_BUF` buffer size.

## ■ Errors

If this function returns `<0` to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <gcLib.h>
#include <gcerr.h>
#include <gcisdn.h>

int set_parm(LINEDEV ldev)
{
    GC_INFO gc_error_info; /* GlobalCall error information data */
    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_ErrorInfo( &gc_error_info );
        printf ("Error: gc_SetParm() on device handle: 0x%x, GC ErrorValue: 0x%x - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%x - %s\n",
                ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    return (0);
}
```

*gc\_SetParm()* — *set the default parameters*



■ **See Also**

- [gc\\_GetParm\(\)](#)

## `gc_SetUpTransfer()`

**Name:** `int gc_SetUpTransfer(calltohold, consultationcall, mode)`

**Inputs:**

<code>CRN calltohold</code>	• call reference number for call to be transferred
<code>CRN *consultationcall</code>	• pointer to call reference number returned for the consultation call
<code>unsigned long mode</code>	• async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** advanced call model

**Mode:** asynchronous or synchronous

**Platform and Technology:** Springware: T-1/E-1 (PDKRT only)†

†See the *Global Call Technology User's Guide* for additional information.

### ■ Description

The `gc_SetUpTransfer()` function initiates a supervised call transfer and allocates a call reference number (CRN) for the consultation call. A supervised transfer means that the party transferring the call consults with the party receiving the transferred call, before the transfer is completed.

The CRN allocated by the `gc_SetUpTransfer()` function is known as the consultation CRN. The consultation CRN is the same as the CRN returned by the `gc_MakeCall()` function when the consultation call is established. The consultation CRN is used by the `gc_CompleteTransfer()` function to complete the transfer request. The `gc_SwapHold()` function also uses the consultation CRN when switching between the call on hold (the call pending transfer) and the active call.

For more information about supervised call transfers, see the information about the advanced call model in the *Global Call API Programming Guide*.

Parameter	Description
<b>calltohold</b>	call reference number for the existing call to be transferred
<b>consultationcall</b>	points to the memory location where the call reference number for the consultation call is to be stored
<b>mode</b>	set to <code>EV_ASYNC</code> for asynchronous execution or to <code>EV_SYNC</code> for synchronous execution

### ■ Termination Events

`GCEV_SETUPTRANSFER`

indicates that the function was successful, that is, the supervised call transfer was initiated.

**GCEV\_TASKFAIL**

indicates that the function failed. For more information, see the “Error Handling” section in the *Global Call API Programming Guide*.

■ **Cautions**

The **gc\_SetUpTransfer( )** function can be used only when an active call is in the Connected call state.

■ **Errors**

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function for error information. If the GCEV\_TASKFAIL event is received, use the **gc\_ResultInfo( )** function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

■ **Example**

```
#include <stdio.h>
#include <srllib.h>
#include <gcclib.h>
#include <gcerr.h>

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

/*
 * Data structure which stores all information for each line
 */
static struct linebag {
    LINEDEV    ldev;          /* GlobalCall API line device handle */
    CRN        original_crn;   /* GlobalCall API call handle */
    CRN        consultation_crn; /* GlobalCall API call handle */
    int        blocked;        /* channel blocked/unblocked */
    int        networkh;       /* network handle */
} port[MAX_CHAN+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".
 * 3. A call has been established (original_crn) and is in connected state.
 */
int call_setuptransfer(int port_num)
{
    GC_INFO gc_error_info;    /* GlobalCall error information data */

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

    /* Put the original call on hold and allocate a CRN for the consultation call */

    if (gc_SetupTransfer(pline->original_crn, &pline->consultation_crn, EV_ASYNC) == -1)
    {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_SetupTransfer() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s",

```



```

        CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
        pline->ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
        gc_error_info.ccLibId, gc_error_info.ccLibName,
        gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}
/* The gc_MakeCall() function can be called here to establish the consultation call. */
/* The consultation_crn can be re-used/over-written here since the CRN will be the same. */
return (0);
}

```

#### ■ See Also

- [gc\\_CompleteTransfer\(\)](#)
- [gc\\_BlindTransfer\(\)](#)
- [gc\\_SwapHold\(\)](#)

## gc\_SetUserInfo( )

**Name:** int gc\_SetUserInfo (target\_type, target\_id, infoparmblkp, duration)

**Inputs:**

int target_type	• type of target object (line device or call reference number)
long target_id	• ID of target: either line device handle or call reference number
GC_PARM_BLK infoparmblkp	• pointer to user information GC_PARM_BLK as specified by the technology
int duration	• duration of the user information setting

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h

**Category:** FTE

**Mode:** synchronous

**Platform and** Springware: ISDN†

**Technology:** DM3: ISDN†

IP (host-based stack† and embedded stack†)

†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The **gc\_SetUserInfo( )** function permits the setting of technology-specific user information. The definition and use of the user information is totally technology dependent. See the associated *Global Call Technology User's Guide* for definition and usage of the user information parameter.

For the **gc\_SetUserInfo( )** function, the target type identifies whether the basic entity is either a line device (GCTGT\_GCLIB\_CHAN) or a call (GCTGT\_GCLIB\_CRN). See [Section 6.2, “Target Objects”](#), on page 417 for more information.

Parameter	Description
<b>target_type</b>	target object type. Valid values are: <ul style="list-style-type: none"> <li>• GCTGT_GCLIB_CHAN</li> <li>• GCTGT_GCLIB_CRN</li> </ul> See <a href="#">Table 22, “GC Parameter Entry List Maintained in GCLIB”</a> , on page 422 for details.
<b>target_id</b>	target object identifier. This identifier, along with target_type, uniquely specifies the target object.
<b>infoparmblkp</b>	points to <a href="#">GC_PARM_BLK</a> which contains parameters defining user information. See the appropriate <i>Global Call Technology User's Guide</i> for GC_PARM_BLK pointer referenced structure definitions.

Parameter	Description
<b>duration</b>	duration of the user information block setting: <ul style="list-style-type: none"> <li>GC_SINGLECALL – effective for the next call if in the Idle or Null call state, otherwise effective for the current call. The user information returns to its default when the call is cleared, and the line returns to Idle state.</li> <li>GC_ALLCALLS – effective for the current or next call, and all subsequent calls.</li> </ul>

## ■ Termination Events

None

## ■ Cautions

If the duration is specified as GC\_ALLCALLS, the specified user information block is maintained for either the duration of the line device (that is, until **gc\_Close()** is invoked on the device) or until **gc\_SetUserInfo()** is invoked again. In the case where **gc\_SetUserInfo()** is invoked subsequent times on a single line device, each prior data setting is overwritten with the data specified in each subsequent call.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
#include "gclib.h"
#include "gcerr.h"
#include "gcisdn.h"

/* ISDN Example: Include specified information element in the next outbound message. */
/* Refer to "ISDN Technology User's Guide" for details. */
int SetInfoElemt(LINEDEV linedev, IE_BLK *ie_blkp)
{
    GC_PARM_BLKP infoparmblkp = NULL; /* input parameter block pointer */
    int retval = GC_SUCCESS;
    GC_INFO gc_error_info; /* GlobalCall error information data */
    GC_PARM_DATAP t_parm_datap = NULL;

    /* allocate GC_PARM_BLK for call progress message parameter */
    gc_util_insert_parm_ref(&infoparmblkp, GCIS_SET_IE, GCIS_PARM_UIEDATA,
        sizeof(IE_BLK), ie_blkp);

    if (infoparmblkp == NULL)
    {
        /* memory allocation error */
        return(-1);
    }
}
```

```
/* configure user specified info element for transmitting in next outbound message
   to network */
retval = gc_SetUserInfo(GCTGT_GCLIB_NETIF, linedev, infoparmblkp, GC_SINGLECALL);

if (retval != GC_SUCCESS)
{
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_SetUsrInfo() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
           CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
           linedev, gc_error_info.gcValue, gc_error_info.gcMsg,
           gc_error_info.ccLibId, gc_error_info.ccLibName,
           gc_error_info.ccValue, gc_error_info.ccMsg);
}

/* free the parameter block */
gc_util_delete_parm_blk(infoparmblkp);

return (retval);
}
```

**■ See Also**

- [gc\\_GetUserInfo\(\)](#)

## `gc_SetUsrAttr()`

**Name:** `int gc_SetUsrAttr(linedev, usrattr)`

**Inputs:** `LINEDEV linedev`      • Global Call line device handle  
`void *usrattr`      • user attribute

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** system controls and tools

**Mode:** synchronous

**Platform and Technology:** All

---

### ■ Description

The `gc_SetUsrAttr()` function sets an attribute defined by the user. For example, the `usrattr` parameter can be used 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 or line device identification. The attribute number is retrieved using the `gc_GetUsrAttr()` function.

Parameter	Description
<code>linedev</code>	Global Call line device handle
<code>usrattr</code>	user defined attribute. Applications can recall this value by calling <code>gc_GetUsrAttr()</code> .

### ■ Termination Events

None

### ■ Cautions

None

### ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

**■ Example**

```
#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)
{
    GC_INFO gc_error_info; /* GlobalCall error information data */

    /*
     * 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_ErrorInfo( &gc_error_info );
        printf ("Error: gc_SetUsrAttr() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    return (0);
}
```

**■ See Also**

- [gc\\_GetUsrAttr\(\)](#)
- [gc\\_OpenEx\(\)](#)

## `gc_SndFrame()`

**Name:** `int gc_SndFrame(linedev, l2_blkp)`

**Inputs:** `LINEDEV linedev` • line device handle for the D channel  
`GC_L2_BLK *l2_blkp` • pointer to location where transmit frame is stored

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** interface specific (deprecated)

**Mode:** synchronous

**Platform and** Springware: ISDN†

**Technology:** DM3: ISDN†

†See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

**Note:** The `gc_SndFrame()` function is deprecated in this software release. The suggested equivalent is `gc_Extension()`.

The `gc_SndFrame()` function sends a Layer 2 frame to the ISDN data link layer. When the data link layer (Layer 2 access) is successfully established, a `GCEV_D_CHAN_STATUS` event is sent to the application. If the data link layer is not established before the function is called, the function fails.

**Note:** To enable Layer 2 access, set parameter number 24 to 01 in the firmware parameter file. When Layer 2 access is enabled, only the `gc_GetFrame()` and `gc_SndFrame()` functions can be used (no calls can be made).

Parameter	Description
<code>linedev</code>	Global Call line device handle for the D channel board
<code>l2_blkp</code>	points to the memory location of the buffer containing the requested transmit frame. The transmit frame uses the <code>GC_L2_BLK</code> data structure. See <code>GC_L2_BLK</code> , on page 389 for further information.

### ■ Termination Events

None

### ■ Cautions

The ISDN data link layer must be successfully established before the function is called or the function will fail. To establish the data link layer (Layer 2 access), set parameter 24 to 01 in the firmware parameter file.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <gcLib.h>
#include <gcerr.h>
#include <gcisdn.h>

METAEVENT metaevent;

int send_frame(GC_L2_BLK *sndfrmptr)
{
    LINEDEV    ldev;           /* Line device */
    int         state;
    GC_INFO     gc_error_info;  /* GlobalCall error information data */
    char        devname[] = ":N_dtiB1:P_isdn";

    /*
     * Do the following:
     * 1. Open the D channel
     * 2. Send the frame
     */

    if(gc_OpenEx(&ldev, devname, EV_SYNC, &ldev) < 0) {
        /* Process the error as decided earlier */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_OpenEx() on devname: %s, GC ErrorValue: 0x%x - %s, CCLibID: %i - %s,\n",
                CC ErrorValue: 0x%x - %s\n",
                devname, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (-1);
    }

    if(gc_GetLineDevState(&ldev, GCGLS_DCHANNEL, &state) < 0) {
        /* Process the error as decided earlier */
    }
    return 0;
}

/* Retrieve events from SRL */
int evt_hdlr()
{
    LINEDEV    ldev;           /* Line device */
    GC_L2_BLK  sndfrm;         /* Buffer to store frame to send */
    GC_L2_BLK  *sndfrmptr      /* Pointer to frame to send */
    GC_L2_BLK  recvfrmptr;     /* Buffer to store received frame */
    int        retcode;        /* Return value */

    retcode = gc_GetMetaEvent(&metaevent);
    if (retcode < 0 ) {
        /* get and process the error */
    }

    /* Continue with normal processing */

    ldev = metaevent.evtdev;
```



```

switch(metaevent.evtttype) {

case GCEV_L2FRAME:
    /* retrieve signaling information from queue */
    if ( gc_GetFrame(ldev, &recvfrmprtr) != GC_SUCCESS) {
        /* Process the error as decided earlier */
    }
    else {
        /* succeeded, process signaling information */
    }
    break;

case GCEV_D_CHAN_STATUS:
    /* Send the message */
    /* Fill up the sndfrm according to call control technology being used */
    sndfrmprtr = &sndfrm;
    if(gc_SndFrame(ldev, sndfrmprtr) != GC_SUCCESS) {
        /* Process the error as decided earlier */
    }
    else {
        /* succeeded, process signaling information */
    }
}
return 0;
}

```

#### ■ See Also

- [`gc\_Extension\(\)`](#)
- [`gc\_GetFrame\(\)`](#)
- [`gc\_SetUserInfo\(\)`](#)

**gc\_SndMsg( )****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**Category:** interface specific (deprecated)**Mode:** synchronous**Platform and** Springware: ISDN†**Technology:** DM3: ISDN†

SS7†

†See the *Global Call Technology User's Guides* for additional information.**■ Description****Note:** The **gc\_SndMsg( )** function is deprecated in this software release. The suggested equivalent is [gc\\_Extension\( \)](#).

The **gc\_SndMsg( )** function sends non-call state related ISDN messages 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 call reference number (CRN) assigned to it.See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>linedev</b>	line device number for the time slot level line device (the B channel)
<b>crn</b>	call reference number
<b>msg_type</b>	specifies the type of message to be sent. See the appropriate <i>Global Call Technology User's Guide</i> for details.
<b>sndmsgptr</b>	points to the buffer containing the IEs to be sent in the message. See <a href="#">GC_IE_BLK</a> , on page 387 for data structure details.

**■ Termination Events**

None

## ■ Cautions

- If this function is invoked for an unsupported technology, the function will fail. The error value EGC\_UNSUPPORTED will be the Global Call value returned when the *gc\_ErrorInfo()* function is used to retrieve the error code.
- For some call control libraries (for example, the ISDN library), if an invalid parameter is used for a *gc\_SndMsg()* call, the invalid parameter is ignored, processing continues, and the function terminates normally.

## ■ Errors

If this function returns <0 to indicate failure, use the *gc\_ErrorInfo()* function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ 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) has been
 *    called to convert the event into metaevent.
 * 4. a GCEV_OFFERED has been detected.
 * 5. a call has been established.
 */

#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>

/*
 * the following ISDN info elem block structure can be passed to the function
 * via the GC_IE_BLK structure).
 */
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 ldev;          /* Line device */
    GC_INFO gc_error_info; /* GlobalCall error information data */
}
```

```
if(gc_CRN2LineDev(crn, &ldev) != GC_SUCCESS) {
    /* process error return as shown */
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_CRN2LineDev() on crn: 0x%lx, GC ErrorValue: 0x%hx - %s,
           CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
           crn, gc_error_info.gcValue, gc_error_info.gcMsg,
           gc_error_info.ccLibId, gc_error_info.ccLibName,
           gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}

if(gc_SndMsg(ldev, crn, msg_type, sndmsgp) != GC_SUCCESS) {
    gc_ErrorInfo( &gc_error_info );
    printf ("Error: gc_SndMsg() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
           CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
           ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
           gc_error_info.ccLibId, gc_error_info.ccLibName,
           gc_error_info.ccValue, gc_error_info.ccMsg);
    return (gc_error_info.gcValue);
}

return(0);
}
```

#### ■ See Also

- [gc\\_Close\(\)](#)
- [gc\\_Extension\(\)](#)
- [gc\\_GetCallInfo\(\)](#)
- [gc\\_GetSigInfo\(\)](#)
- [gc\\_SetUserInfo\(\)](#)

## `gc_Start()`

**Name:** `int gc_Start(startp)`

**Inputs:** `GC_START_STRUCT *startp` • pointer to call control library information

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** system controls and tools

**Mode:** synchronous

**Platform and** All†

**Technology:** †See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

The `gc_Start()` function starts and initializes call control libraries. This function **must** be called before any other Global Call function is called. The application can specify one or more call control libraries to be started through the `GC_START_STRUCT` data structure.

To start all the supported call control libraries:

- Pass a NULL parameter; that is, use `gc_Start(NULL)`.
- Set the `num_cclibs` field in the `GC_START_STRUCT` structure pointed to by `startp` to a value of `GC_ALL_LIB`.

The function opens the call control libraries that interface directly to the network interface so that these libraries can be used by the Global Call library.

The `gc_Start()` function returns 0 if the call control libraries have successfully started. If a NULL parameter is passed to `gc_Start()` and a call control library is not present, an error is not returned. However, the state of the missing call control library will be “failed”. If `GC_START_STRUCT` is used, an error is returned if one or more of the libraries fail to load.

Successfully started libraries are available to be used by the Global Call functions and are called “available” libraries. Libraries that fail to start are called “failed” libraries. Use the `gc_CCLibStatusEx()` function to determine the status (available, configured, failed) of one or all call control libraries.

Parameter	Description
<code>startp</code>	points to list of call control libraries to be loaded and started. If NULL, then all of the supported libraries will be started.

### ■ Termination Events

None

## ■ Cautions

- The `gc_Start()` function should be called only once per application execution. Calling `gc_Start()` a second time (i.e., after `gc_Stop()` function) is **not** recommended although it is allowed in a debug environment. If calling `gc_Start()` after `gc_Stop()` is needed in your application, it should be done only a limited number of times, not frequently.
- For Linux applications, the `gc_Start()` function **must** be called from the parent process when creating child processes.
- For multi-threaded 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.

## ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file.

## ■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <gcilib.h>
#include <gcerr.h>

int sysinit()
{
    GC_START_STRUCT gcilib_start;    /* Structure for gc_Start() */
    GC_INFO          gc_error_info;  /* GlobalCall error information data */

    GC_CUSTOMLIB_STRUCT custom_lib1 = {
        "libgcxyz.dll",
        "xyz_Start"
    };

    CCLIB_START_STRUCT cclib_start[]={
        {"GC_ANAPI_LIB",NULL},
        {"GC_ICAPI_LIB",NULL},
        {"GC_PDKRT_LIB",NULL},
        {"GC_CUSTOM1_LIB", (void *)&custom_lib1},
    };

    gcilib_start.num_cclibs = 4;
    gcilib_start.cclib_list = cclib_start;

    /* Next issue a gc_Start() Call */
    if ( gc_Start( &gcilib_start ) != GC_SUCCESS )    {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_Start(), GC ErrorValue: 0x%xh - %s, CCLibID: %i - %s,\n",
            GC_ErrorValue: 0x%xh - %s\n",
            gc_error_info.gcValue, gc_error_info.gcMsg,
            gc_error_info.ccLibId, gc_error_info.ccLibName,
            gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    return (0);

    /* Next open the GlobalCall Line Devices */
}
```

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

■ **See Also**

- [gc\\_CCLibStatusEx\(\)](#)
- [gc\\_Stop\(\)](#)

## gc\_StartTrace( )

**Name:** int gc\_StartTrace(linedev, tracefilename)

**Inputs:** LINEDEV linedev      • Global Call line device handle  
char \*tracefilename      • pointer to file name for trace

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** interface specific

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1 (PDKRT only)†, ISDN†, Analog (PDKRT only)†

DM3: ISDN†  
SS7†

†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The **gc\_StartTrace( )** function starts the logging of debug information for a specified line device. The trace continues until a **gc\_StopTrace( )** function is issued. Complete information about the trace is not available until the **gc\_StopTrace( )** function is executed.

For information about using the **gc\_StartTrace( )** function with a specific technology, see the appropriate *Global Call Technology User's Guide*.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>tracefilename</b>	points to file name for the trace

### ■ Termination Events

None

### ■ Cautions

If this function is invoked for an unsupported technology, the function will fail. The error value EGC\_UNSUPPORTED will be the Global Call value returned when the **gc\_ErrorInfo( )** function is used to retrieve the error code.

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*.



All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

### ■ Example

```
#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 */
char   devname[] = ":N_dtiB1:P_ISDN";
GC_INFO gc_error_info; /* GlobalCall error information data */

main()
{
    if(gc_OpenEx(&bdev, devname, EV_SYNC, &ldev) != GC_SUCCESS) {
        /* Process the error as decided earlier */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_OpenEx() on devname: %s, GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s,
                CC ErrorValue: 0x%lx - %s\n",
                devname, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }

    /* 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 */
    }
    .
    .
    .
}
```

### ■ See Also

- [gc\\_StopTrace\(\)](#)

## gc\_Stop()

**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

**Platform and** All†

**Technology:** †See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

The **gc\_Stop()** function stops call control libraries and releases resources. This function **must** be the last Global Call function called before exiting the application.

For Linux applications, the **gc\_Stop()** function must be called from the parent process when child processes are used.

For multi-threaded applications, the **gc\_Stop()** function must be called from the same thread that issued the **gc\_Start()** call.

### ■ Termination Events

None

### ■ Cautions

- The **gc\_Stop()** function must be called before exiting the application.
- All open devices should be closed before issuing a **gc\_Stop()** function.
- No Global Call functions should be called after **gc\_Stop()**. Calling **gc\_Start()** a second time after **gc\_Stop()** is **not** recommended although it is allowed in a debug environment. If calling **gc\_Start()** after **gc\_Stop()** is needed in your application, it should be done only a limited number of times, not frequently.

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
#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[] */
    GC_INFO      gc_error_info;   /* GlobalCall error information data */

    /* 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_ErrorInfo( &gc_error_info );
        printf ("Error: gc_Stop(), GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s,
                CC ErrorValue: 0x%lx - %s\n",
                gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
    }

    /* Close all open file handles corresponding to recorded files and exit */
    exit(exit_code);
}
```

## ■ See Also

- [`gc\_Start\(\)`](#)

## gc\_StopTrace( )

**Name:** int gc\_StopTrace(linedev)

**Inputs:** LINEDEV linedev • Global Call line device handle

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** interface specific

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1 (PDKRT only), ISDN†, Analog (PDKRT only)

DM3: ISDN†  
SS7†

†See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The **gc\_StopTrace( )** function stops the logging of debug information that was started using the **gc\_StartTrace( )** function and closes the trace file.

Parameter	Description
linedev	Global Call line device handle

### ■ Termination Events

None

### ■ Cautions

If this function is invoked for an unsupported technology, the function fails. The error value EGC\_UNSUPPORTED will be the Global Call value returned when the **gc\_ErrorInfo( )** function is used to retrieve the error code.

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
#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 */
GC_INFO gc_error_info; /* GlobalCall error information data */

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 */
    }
    .
    .
    .
}
```

## ■ See Also

- [gc\\_StartTrace\(\)](#)

## gc\_StopTransmitAlarms( )

**Name:** int gc\_StopTransmitAlarms(linedev, aso\_id, alarm\_list, mode)

**Inputs:**

LINEDEV linedev	• line device
unsigned long aso_id	• alarm source object ID
ALARM_LIST *alarm_list	• pointer to the alarm list
unsigned long mode	• sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** GCAMS

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN

DM3: T-1/E-1, ISDN

### ■ Description

The **gc\_StopTransmitAlarms( )** function stops the transmission of one or more alarms to the remote side. To stop the transmission, the alarms must have been initiated by the **gc\_TransmitAlarms( )** function. The function stops transmitting all alarms specified in **alarm\_list** for the given line device.

Parameter	Description
<b>linedev</b>	Global Call line device
<b>aso_id</b>	alarm source object (ASO) ID. Use the <b>gc_AlarmSourceObjectNameToID( )</b> function to obtain the ASO ID for the desired alarm source object. Use the <b>gc_GetAlarmSourceObjectNetworkID( )</b> function to obtain the network ID.  ALARM_SOURCE_ID_NETWORK_ID can be used if the network ASO ID associated with the line device is desired.  For a list of ASO IDs that are known to Global Call, see <a href="#">Table 18, “Alarm Source Object IDs”</a> , on page 417.
<b>alarm_list</b>	points to the alarm list. See <a href="#">ALARM_LIST</a> , on page 368 for descriptions of the fields in the ALARM_LIST data structure.  <b>Note:</b> Only the alarm_number field of ALARM_FIELD in the ALARM_LIST data structure is used.
<b>mode</b>	set to EV_SYNC for synchronous mode (only synchronous mode is supported)

## ■ Termination Events

None

## ■ Cautions

- Detailed knowledge of the alarm source object is required to use this function properly.
- If any of the alarms failed to stop transmitting, the function will return an error. To ensure that an error is received for every failure, stop the alarms one at a time.
- The list of possible alarms used by the ASO is defined by the ASO itself, not by Global Call. See the *Global Call API Programming Guide* for more information about ASOs.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

## ■ Example

```
#include <stdio.h>
#include <stdlib.h>
#include <gclib.h>
#include <gcerr.h>
#include <dtilib.h> /* for ASO symbols */

int rc;
LINEDEV linedev;
ALARM_LIST alarm_list;
GC_INFO gc_error_info; /* GlobalCall error information data */

/*
-- This code assumes that linedev is already assigned
-- it also assumes that linedev's alarm source object
-- is known to be Springware E1
-- this could have been done via gc_GetAlarmSourceObjectNetworkID
*/
/* init all to 0 */
memset(&alarm_list, '\0', sizeof(ALARM_LIST));

/* stop transmitting one alarm - DTE1_RSA1 */
alarm_list.n_alarms = 1;
alarm_list.alarm_fields[0].alarm_number = DTE1_RSA1;

rc = gc_StopTransmitAlarms(linedev, ALARM_SOURCE_ID_SPRINGWARE_E1,
                           &alarm_list, EV_SYNC);

if (rc < 0)
{
/* get and process the error */
gc_ErrorInfo(&gc_error_info);
printf ("Error: gc_StopTransmitAlarms() on device handle: 0x%lx,
GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
linedev, gc_error_info.gcValue, gc_error_info.gcMsg,
gc_error_info.ccLibId, gc_error_info.ccLibName,
gc_error_info.ccValue, gc_error_info.ccMsg);
return (gc_error_info.gcValue);
}
```

■ See Also

- [gc\\_TransmitAlarms\(\)](#)



## gc\_SwapHold()

**Name:** int gc\_SwapHold(activecall, callonhold, mode)

**Inputs:**

CRN activecall	• call reference number for active call
CRN *callonhold	• pointer to call reference number for call on hold (pending transfer)
unsigned long mode	• async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** gcLib.h  
gcErr.h

**Category:** advanced call model

**Mode:** asynchronous or synchronous

**Platform and Technology:** Springware: T-1/E-1 (PDKRT only)†

†See the *Global Call Technology User's Guides* for additional information.

---

### ■ Description

The **gc\_SwapHold()** function is used to switch between an active call and a call on hold. The **gc\_SwapHold()** function can also be used in a supervised transfer to switch between the consultation call and the call pending transfer. This function can be called only when a consultation call has been established (see the **gc\_CompleteTransfer()** function description).

For more information about supervised transfers, see the information about the advanced call model in the *Global Call API Programming Guide*.

Parameter	Description
<b>activecall</b>	call reference number for the original (active) call
<b>callonhold</b>	points to call reference number for the call on hold/pending transfer
<b>mode</b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

### ■ Termination Events

GCEV\_SWAPHOLD

indicates the successful completion of the **gc\_SwapHold()** function, that is, the active call and the call on hold (pending transfer) were switched.

GCEV\_TASKFAIL

indicates that the function failed. For more information, see the “Error Handling” section in the *Global Call API Programming Guide*.

## ■ Cautions

The **gc\_SwapHold()** function can be called only when one call is the Connected state and the other call is in the OnHoldPendingTransfer state or the OnHold state.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function for error information. If the GCEV\_TASKFAIL event is received, use the **gc\_ResultInfo()** function to retrieve information about the event. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdner.h* file for the ISDN call control library).

## ■ Example

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

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

/*
 * Data structure which stores all information for each line
 */
static struct linebag {
    LINEDEV    ldev;          /* GlobalCall API line device handle */
    CRN         original_crn;  /* GlobalCall API call handle */
    CRN         consultation_crn; /* GlobalCall API call handle */
    int         blocked;      /* channel blocked/unblocked */
    int         networkh;     /* network handle */
} port[MAX_CHAN+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".
 * 3. A call has been established (original_crn) and
 *    is in connected state.
 * 4. The gc_SetUpTransfer() has been called successfully
 *    to initiate the transfer.
 * 5. A consultation call has also been established.
 */
int call_swaphold(int port_num)
{
    GC_INFO gc_error_info;    /* GlobalCall error information data */

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

    /* Put the consultation call on hold, and make the original call active */
    if (gc_SwapHold(pline->original_crn, pline->consultation_crn, EV_ASYNC) == -1) {
        /* process error return as shown */
        gc_ErrorInfo(&gc_error_info);
        printf ("Error: gc_SwapHold() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                pline->original_crn, pline->consultation_crn, gc_error_info.error_text,
                gc_error_info.cclib_id, gc_error_info.cclib_text, gc_error_info.error_value,
                gc_error_info.cc_error_value);
    }
}
```

```
        pline->ldev, gc_error_info.gcValue, gc_error_info.gcMsg,  
        gc_error_info.ccLibId, gc_error_info.ccLibName,  
        gc_error_info.ccValue, gc_error_info.ccMsg);  
    return (gc_error_info.gcValue);  
}  
  
return (0);  
}
```

#### ■ See Also

- [gc\\_CompleteTransfer\(\)](#)
- [gc\\_BlindTransfer\(\)](#)
- [gc\\_SetUpTransfer\(\)](#)

## gc\_TransmitAlarms( )

**Name:** int gc\_TransmitAlarms(linedev, aso\_id, alarm\_list, mode)

**Inputs:**

LINEDEV linedev	• line device
unsigned long aso_id	• alarm source object ID
ALARM_LIST *alarm_list	• pointer to the alarm list
unsigned long mode	• sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** GCAMS

**Mode:** synchronous

**Platform and Technology:** Springware: T-1/E-1, ISDN

DM3: T-1/E-1, ISDN

### ■ Description

The **gc\_TransmitAlarms( )** function starts the transmission of alarms to the remote side. The function starts transmitting all alarms specified in **alarm\_list** for the given line device.

Parameter	Description
<b>linedev</b>	line device
<b>aso_id</b>	alarm source object (ASO) ID. Use the <a href="#">gc_AlarmSourceObjectNameToID( )</a> function to obtain the ASO ID for the desired alarm source object. Use the <a href="#">gc_GetAlarmSourceObjectNetworkID( )</a> function to obtain the network ID.  ALARM_SOURCE_ID_NETWORK_ID can be used if the network ASO ID associated with the line device is desired.  For a list of ASO IDs that are known to Global Call, see <a href="#">Table 18, “Alarm Source Object IDs”</a> , on page 417.
<b>alarm_list</b>	points to the alarm list. See <a href="#">ALARM_LIST</a> , on page 368 for descriptions of the fields in the ALARM_LIST data structure.  <b>Note:</b> Only the alarm_number field of ALARM_FIELD in the ALARM_LIST data structure is used.
<b>mode</b>	set to EV_SYNC for synchronous mode (only synchronous mode is supported)

## ■ Termination Events

None

## ■ Cautions

- Detailed knowledge of the alarm source object is required to use this function properly.
- If any of the alarms failed to start transmitting, the function will return an error. To ensure that an error is received for every failure, start the alarms one at a time.
- If the function is partially successful, that is, if only some of the alarms in **alarm\_list** start transmitting, the alarms that were successfully started will not be stopped.
- The list of possible alarms used by the ASO is defined by the ASO itself, not by Global Call. See [Section 6.1, “Alarm Source Object IDs”](#), on page 417 and the *Global Call API Programming Guide* for more information about ASOs.
- Some alarm source objects may require that a `gc_SetAlarmParm()` be issued before calling this function. This requirement is alarm source object dependent.

## ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

```
#include <stdio.h>
#include <stdlib.h>
#include <gclib.h>
#include <gcerr.h>
#include <dtlib.h>                /* for ASO symbols */

int          rc;
LINEDEV      linedev;
ALARM_LIST   alarm_list;
GC_INFO      gc_error_info; /* GlobalCall error information data */

/*
-- This code assumes that linedev is already assigned
-- it also assumes that linedev's alarm source object
-- is known to be Springware E1
-- this could have been done via gc_GetAlarmSourceObjectNetworkID
*/

/* init all to 0 */
memset(&alarm_list, '\0', sizeof(ALARM_LIST));

/* transmit one alarm - DTE1_RSA1 */
alarm_list.n_alarms = 1;
alarm_list.alarm_fields[0].alarm_number = DTE1_RSA1;

rc = gc_TransmitAlarms(linedev, ALARM_SOURCE_ID_SPRINGWARE_E1, &alarm_list, EV_SYNC);
```

```
if (rc < 0)
{
    /* get and process the error */
}
```

■ **See Also**

- [gc\\_StopTransmitAlarms\(\)](#)

## `gc_UnListen()`

**Name:** `int gc_UnListen(linedev, mode)`

**Inputs:** `LINEDEV linedev`      • Global Call line device handle  
          `unsigned long mode`      • `async` or `sync`

**Returns:** 0 if successful  
          <0 if failure

**Includes:** `gclib.h`  
          `gcerr.h`

**Category:** system controls and tools

**Mode:** asynchronous or synchronous

**Platform and** Springware: T-1/E-1 (PDKRT only)

**Technology:** DM3: T-1/E-1, ISDN, Analog  
                  SS7

                  IP (host-based stack† and embedded stack†)

                  †See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The `gc_UnListen()` function disconnects a channel from the network CT Bus time slot for a line device that was connected previously using the `gc_Listen()` function.

Parameter	Description
<code>linedev</code>	Global Call line device handle
<code>mode</code>	set to <code>EV_ASYNC</code> for asynchronous execution or to <code>EV_SYNC</code> for synchronous execution

### ■ Termination Events

`GCEV_UNLISTEN`  
          indicates that the time slot has been unrouted successfully.

`GCEV_TASKFAIL`  
          indicates that the time slot unrouting has failed.

### ■ Cautions

For routing on resources such as voice resources, the routing function for the corresponding library must be called, for example, `dx_getxmitslot()`, `dx_listen()`, `dx_unlisten()`.

### ■ Errors

If this function returns <0 to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*.

All Global Call error codes are defined in the *gcerr.h* file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, *ccerr.h* or *isdnerr.h* file for the ISDN call control library).

### ■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <gcLib.h>
#include <gcerr.h>

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

/*
 * Data structure which stores all information for each line
 */
static struct linebag {
    LINEDEV  ldev;          /* GlobalCall API line device handle */
    CRN      crn;           /* GlobalCall API call handle */
    int      blocked;       /* channel blocked/unblocked */
    int      networkh;      /* network handle */
} port[MAX_CHAN+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 and voice handle is stored in linebag structure "port"
 */
int call_unlisten(int port_num)
{
    GC_INFO      gc_error_info;  /* GlobalCall error information data */

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

    /* Disconnect receive of digital board 1, timeslot 1
     from all CTbus timeslots */
    if (gc_UnListen(pline->ldev, EV_SYNC) == -1)
    {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_UnListen() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                pline->ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
        return (gc_error_info.gcValue);
    }
    return (0);
}
```

### ■ See Also

- [gc\\_GetCTInfo\(\)](#)
- [gc\\_GetXmitSlot\(\)](#)
- [gc\\_Listen\(\)](#)



## gc\_util\_delete\_parm\_blk( )

**Name:** void gc\_util\_delete\_parm\_blk(parm\_blk )

**Inputs:** GC\_PARM\_BLK\* parm\_blk • pointer to [GC\\_PARM\\_BLK](#) to be deleted

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** GC\_PARM\_BLK utility

**Mode:** synchronous

**Platform and Technology:** All

### ■ Description

The **gc\_util\_delete\_parm\_blk( )** function deletes the specified [GC\\_PARM\\_BLK](#).

Parameter	Description
<b>parm_blk</b>	points to the GC_PARM_BLK to be deleted. For more information about the GC_PARM_BLK data structure, see <a href="#">GC_PARM_BLK</a> , on page 392.

### ■ Termination Events

None

### ■ Cautions

None

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo( )** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file.

### ■ Example

```
#include "gclib.h"

void main( )
{
    GC_PARM_BLK my_blkp = NULL;
    GC_PARM_DATAP my_parmp;
    int type = 1;
```

```
/* insert parm by reference */
if ( gc_util_insert_parm_ref( &my_blkp, GC_SET_SERVREQ, PARM_REQTYPE,
    sizeof( int ), &type ) != GC_SUCCESS )
{
    /* Process error */
}

/* insert parm by value */
if ( gc_util_insert_parm_val( &my_blkp, GC_SET_SERVREQ, PARM_ACK,
    sizeof( short ), GC_ACK ) != GC_SUCCESS )
{
    /* Process error */
}

/* Now we should have a GC_PARM_BLK with 2 parameters */

/* Following use of gc_util_next_parm retrieves the first parameter in a
 * GC_PARM_BLK, which in this case is PARM_REQTYPE */
my_parmp = gc_util_next_parm( my_blkp, NULL );

/* Retrieve the next parameter after getting the first one */
my_parmp = gc_util_next_parm( my_blkp, my_parmp );

/* This function finds and returns specified parameter, NULL if not found */
my_parmp = gc_util_find_parm( my_blkp, GC_SET_SERVREQ, PARM_ACK );

/* After GC_PARM_BLK is no longer needed, delete the block */
gc_util_delete_parm_blk( my_blkp );

/* Set my_blkp to NULL now that the block has been deleted */
my_blkp = NULL;
}
```

#### ■ See Also

- [gc\\_util\\_insert\\_parm\\_ref\(\)](#)
- [gc\\_util\\_insert\\_parm\\_val\(\)](#)

## gc\_util\_find\_parm( )

**Name:** GC\_PARM\_DATAP gc\_util\_find\_parm(parm\_blk, setID, parmID )

**Inputs:** GC\_PARM\_BLK \*parm\_blk

- pointer to [GC\\_PARM\\_BLK](#) where parameter is possibly located

unsigned short setID

- set ID of parameter to be found

unsigned short parmID

- parm ID of parameter to be found

**Returns:** a pointer to GC\_PARM\_DATAP if successful  
NULL if the parameter is not found in the given [GC\\_PARM\\_BLK](#)

**Includes:** gclib.h  
gcerr.h

**Category:** GC\_PARM\_BLK utility

**Mode:** synchronous

**Platform and Technology:** All

### ■ Description

The **gc\_util\_find\_parm()** function is used to find a parameter in a [GC\\_PARM\\_BLK](#). The **gc\_util\_find\_parm()** function can be used to verify that a particular parameter exists, or to retrieve a particular parameter, or both. The function returns a pointer to the parameter if the parameter exists, or NULL if the parameter is not found in the specified GC\_PARM\_BLK.

Parameter	Description
<b>parm_blk</b>	points to a valid <a href="#">GC_PARM_BLK</a> where the parameter is possibly located
<b>setID</b>	set ID of the parameter to be found
<b>parmID</b>	parm ID of the parameter to be found

### ■ Termination Events

None

### ■ Cautions

- If the same parameter is included twice inside the [GC\\_PARM\\_BLK](#), **only** the first instance of the parameter is returned.
- The pointer returned by **gc\_util\_find\_parm()** can be used to modify that specific parameter within the [GC\\_PARM\\_BLK](#). When the GC\_PARM\_BLK is modified, the changes will not take effect until the appropriate function, such as [gc\\_SetConfigData\(\)](#), is called. The size of the data must remain the same.

## ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file.

## ■ Example

```
#include "gclib.h"

void main( )
{
    GC_PARM_BLK my_blkp = NULL;
    GC_PARM_DATAP my_parmp;
    int type = 1;

    /* insert parm by reference */
    if ( gc_util_insert_parm_ref( &my_blkp, GC_SET_SERVREQ, PARM_REQTYPE,
        sizeof( int ), &type ) != GC_SUCCESS )
    {
        /* Process error */
    }

    /* insert parm by value */
    if ( gc_util_insert_parm_val( &my_blkp, GC_SET_SERVREQ, PARM_ACK,
        sizeof( short ), GC_ACK ) != GC_SUCCESS )
    {
        /* Process error */
    }

    /* Now we should have a GC_PARM_BLK with 2 parameters */

    /* Following use of gc_util_next_parm retrieves the first parameter in a
     * GC_PARM_BLK, which in this case is PARM_REQTYPE */
    my_parmp = gc_util_next_parm( my_blkp, NULL );

    /* Retrieve the next parameter after getting the first one */
    my_parmp = gc_util_next_parm( my_blkp, my_parmp );

    /* This function finds and returns specified parameter, NULL if not found */
    my_parmp = gc_util_find_parm( my_blkp, GC_SET_SERVREQ, PARM_ACK );

    /* After GC_PARM_BLK is no longer needed, delete the block */
    gc_util_delete_parm_blk( my_blkp );

    /* Set my_blkp to NULL now that the block has been deleted */
    my_blkp = NULL;
}
```

## ■ See Also

- [gc\\_util\\_insert\\_parm\\_ref\(\)](#)
- [gc\\_util\\_insert\\_parm\\_val\(\)](#)
- [gc\\_util\\_next\\_parm\(\)](#)
- [gc\\_util\\_delete\\_parm\\_blk\(\)](#)

## *gc\_util\_insert\_parm\_ref()*

**Name:** `int gc_util_insert_parm_ref(parm_blkpp, setID, parmID, data_size, datap)`

**Inputs:** `GC_PARM_BLKP **parm_blkpp` • pointer to the address of a valid [GC\\_PARM\\_BLK](#)  
`unsigned short setID` • set ID of parameter to be inserted  
`unsigned short parmID` • parm ID of parameter to be inserted  
`unsigned char data_size` • size of the data in bytes  
`void *datap` • pointer to the parameter data

**Returns:** 0 if successful  
 <0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** *GC\_PARM\_BLK* utility

**Mode:** synchronous

**Platform and Technology:** All

### ■ Description

The *gc\_util\_insert\_parm\_ref()* function must be used to insert a parameter by reference into a [GC\\_PARM\\_BLK](#). A new *GC\_PARM\_BLK* can be created by inserting the first parameter into the block and setting *\*parm\_blkpp* to NULL. A parameter can be inserted in an existing *GC\_PARM\_BLK* by setting *\*parm\_blkpp* to the address of that block.

**Note:** When the parameters in the *GC\_PARM\_BLK* are retrieved via the [gc\\_util\\_next\\_parm\(\)](#) function, they are retrieved in the same order in which they were inserted.

Parameter	Description
<b>parm_blkpp</b>	points to the address of a valid <a href="#">GC_PARM_BLK</a> where the parameter is to be inserted. Set <i>*parm_blkpp</i> to NULL if the parameter is to be inserted into a new block.
<b>setID</b>	set ID of the parameter to be inserted
<b>parmID</b>	parm ID of the parameter to be inserted
<b>data_size</b>	size, in bytes, of the data associated with this parameter
<b>datap</b>	points to the data associated with this parameter

### ■ Termination Events

None

## ■ Cautions

- The `gc_util_insert_parm_ref()` function does not check whether the set ID or the parm ID is valid, nor does it check for duplicate parameters.
- To avoid a memory leak, any `GC_PARM_BLK` created must be deleted using the `gc_util_delete_parm_blk()` function.
- The maximum size of a `GC_PARM_BLK` is defined as `PARM_BLK_HUGE` bytes. If the maximum size is reached when inserting a parameter, the function will return an `EGC_NOMEM` error.

## ■ Errors

If this function returns `<0` to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file.

## ■ Example

```
#include "gclib.h"

void main( )
{
    GC_PARM_BLK my_blkp = NULL;
    GC_PARM_DATAP my_parmp;
    int type = 1;

    /* insert parm by reference */
    if ( gc_util_insert_parm_ref( &my_blkp, GC_SET_SERVREQ, PARM_REQTYPE,
        sizeof( int ), &type ) != GC_SUCCESS )
    {
        /* Process error */
    }

    /* insert parm by value */
    if ( gc_util_insert_parm_val( &my_blkp, GC_SET_SERVREQ, PARM_ACK,
        sizeof( short ), GC_ACK ) != GC_SUCCESS )
    {
        /* Process error */
    }

    /* Now we should have a GC_PARM_BLK with 2 parameters */

    /* Following use of gc_util_next_parm retrieves the first parameter in a
     * GC_PARM_BLK, which in this case is PARM_REQTYPE */
    my_parmp = gc_util_next_parm( my_blkp, NULL );

    /* Retrieve the next parameter after getting the first one */
    my_parmp = gc_util_next_parm( my_blkp, my_parmp );

    /* This function finds and returns specified parameter, NULL if not found */
    my_parmp = gc_util_find_parm( my_blkp, GC_SET_SERVREQ, PARM_ACK );

    /* After GC_PARM_BLK is no longer needed, delete the block */
    gc_util_delete_parm_blk( my_blkp );

    /* Set my_blkp to NULL now that the block has been deleted */
    my_blkp = NULL;
}
```



*insert a parameter by reference into a GC\_PARM\_BLK — `gc_util_insert_parm_ref()`*

■ **See Also**

- [gc\\_util\\_insert\\_parm\\_val\(\)](#)
- [gc\\_util\\_delete\\_parm\\_blk\(\)](#)

## gc\_util\_insert\_parm\_val( )

**Name:** int gc\_util\_insert\_parm\_val(parm\_blkpp, setID, parmID, data\_size, data)

**Inputs:** GC\_PARM\_BLKP \*\*parm\_blkpp      • pointer to address of a valid [GC\\_PARM\\_BLK](#)  
 unsigned short setID                      • set ID of parameter to be inserted  
 unsigned short parmID                      • parm ID of parameter to be inserted  
 unsigned char data\_size                    • size of the data in bytes  
 unsigned long data                          • actual parameter data

**Returns:** 0 if successful  
<0 if failure

**Includes:** gclib.h  
gcerr.h

**Category:** GC\_PARM\_BLK utility

**Mode:** synchronous

**Platform and Technology:** All

### ■ Description

The **gc\_util\_insert\_parm\_val( )** function must be used to insert a parameter by value into a [GC\\_PARM\\_BLK](#). A new GC\_PARM\_BLK can be created by inserting the first parameter of the block, and setting **\*parm\_blkpp** to NULL. A parameter can be inserted in an existing GC\_PARM\_BLK by setting **\*parm\_blkpp** to the address of that block.

**Note:** When the parameters in the GC\_PARM\_BLK are retrieved via the [gc\\_util\\_next\\_parm\( \)](#) function, they are retrieved in the same order in which they were inserted.

Parameter	Description
<b>parm_blkpp</b>	points to the address of a valid <a href="#">GC_PARM_BLK</a> where the parameter is to be inserted. Set <b>*parm_blkpp</b> to NULL if the parameter is to be inserted into a new block.
<b>setID</b>	set ID of the parameter to be inserted
<b>parmID</b>	parm ID of the parameter to be inserted
<b>data_size</b>	size, in bytes, of the data associated with this parameter
<b>data</b>	actual data associated with this parameter

### ■ Termination Events

None



## ■ Cautions

- The `gc_util_insert_parm_val()` function can only insert data of type char, short, int, or long.
- The `gc_util_insert_parm_val()` function does not check whether the Set ID or the Parm ID is valid, nor does it check for duplicate parameters.
- To avoid a memory leak, any `GC_PARM_BLK` created must be deleted using the `gc_util_delete_parm_blk()` function.
- The maximum size of a `GC_PARM_BLK` is defined as `PARM_BLK_HUGE` bytes. If the maximum size is reached when inserting a parameter the function will return an `EGC_NOMEM` error.

## ■ Errors

If this function returns `<0` to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file.

## ■ Example

```
#include "gclib.h"

void main( )
{
    GC_PARM_BLK my_blkp = NULL;
    GC_PARM_DATAP my_parmp;
    int type = 1;

    /* insert parm by reference */
    if ( gc_util_insert_parm_ref( &my_blkp, GC_SET_SERVREQ, PARM_REQTYPE,
        sizeof( int ), &type ) != GC_SUCCESS )
    {
        /* Process error */
    }

    /* insert parm by value */
    if ( gc_util_insert_parm_val( &my_blkp, GC_SET_SERVREQ, PARM_ACK,
        sizeof( short ), GC_ACK ) != GC_SUCCESS )
    {
        /* Process error */
    }

    /* Now we should have a GC_PARM_BLK with 2 parameters */

    /* Following use of gc_util_next_parm retrieves the first parameter in a
     * GC_PARM_BLK, which in this case is PARM_REQTYPE */
    my_parmp = gc_util_next_parm( my_blkp, NULL );

    /* Retrieve the next parameter after getting the first one */
    my_parmp = gc_util_next_parm( my_blkp, my_parmp );

    /* This function finds and returns specified parameter, NULL if not found */
    my_parmp = gc_util_find_parm( my_blkp, GC_SET_SERVREQ, PARM_ACK );

    /* After GC_PARM_BLK is no longer needed, delete the block */
    gc_util_delete_parm_blk( my_blkp );

    /* Set my_blkp to NULL now that the block has been deleted */
    my_blkp = NULL;
}
```

■ See Also

- [gc\\_util\\_insert\\_parm\\_ref\(\)](#)
- [gc\\_util\\_delete\\_parm\\_blk\(\)](#)

## gc\_util\_next\_parm( )

**Name:** gc\_util\_next\_parm(parm\_blk, cur\_parm )

**Inputs:** GC\_PARM\_BLK \*parm\_blk      • pointer to GC\_PARM\_BLK  
GC\_PARM\_DATAP \*cur\_parm      • pointer to current parameter

**Returns:** a pointer if successful  
NULL if no more parameters exist

**Includes:** gclib.h  
gcerr.h

**Category:** GC\_PARM\_BLK utility

**Mode:** synchronous

**Platform and  
Technology:** All

---

### ■ Description

The **gc\_util\_next\_parm()** function must be used to retrieve the next parameter in a GC\_PARM\_BLK. The function takes a GC\_PARM\_BLK pointer to the block where the data resides and a GC\_PARM\_DATA pointer returned by a previous call to this function. If the first parameter in the block is desired, NULL can be used. The function locates and returns the pointer to the next parameter, or NULL if no more parameters exist.

Parameter	Description
parm_blk	points to the parameter block where data is stored
cur_parm	points to the current parameter. The pointer is obtained through a previous call to this function. Use NULL if the first parameter is desired.

### ■ Termination Events

None

### ■ Cautions

This function always returns a pointer to the next parameter relative to the given parameters, for example, this function will return the pointer to the third element if given the pointer to the second element. It does not return the pointer to the next element relative to the previous call of this function.

### ■ Errors

If this function returns <0 to indicate failure, use the **gc\_ErrorInfo()** function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the *gcerr.h* file.

**■ Example**

```
#include "gclib.h"

void main( )
{
    GC_PARM_BLK my_blkp = NULL;
    GC_PARM_DATAP my_parmp;
    int type = 1;

    /* insert parm by reference */
    if ( gc_util_insert_parm_ref( &my_blkp, GC_SET_SERVREQ, PARM_REQTYPE,
        sizeof( int ), &type ) != GC_SUCCESS )
    {
        /* Process error */
    }

    /* insert parm by value */
    if ( gc_util_insert_parm_val( &my_blkp, GC_SET_SERVREQ, PARM_ACK,
        sizeof( short ), GC_ACK ) != GC_SUCCESS )
    {
        /* Process error */
    }

    /* Now we should have a GC_PARM_BLK with 2 parameters */

    /* Following use of gc_util_next_parm retrieves the first parameter in a
     * GC_PARM_BLK, which in this case is PARM_REQTYPE */
    my_parmp = gc_util_next_parm( my_blkp, NULL );

    /* Retrieve the next parameter after getting the first one */
    my_parmp = gc_util_next_parm( my_blkp, my_parmp );

    /* This function finds and returns specified parameter, NULL if not found */
    my_parmp = gc_util_find_parm( my_blkp, GC_SET_SERVREQ, PARM_ACK );

    /* After GC_PARM_BLK is no longer needed, delete the block */
    gc_util_delete_parm_blk( my_blkp );

    /* Set my_blkp to NULL now that the block has been deleted */
    my_blkp = NULL;
}
```

**■ See Also**

- [\*\*\*gc\\_util\\_find\\_parm\(\)\*\*\*](#)

## `gc_WaitCall()`

**Name:** `int gc_WaitCall(linedev, crnp, waitcallp, timeout, mode)`

**Inputs:**

<code>LINEDEV linedev</code>	• Global Call line device handle
<code>CRN *crnp</code>	• pointer to CRN
<code>GC_WAITCALL_BLK *waitcallp</code>	• reserved for future use
<code>int timeout</code>	• time-out
<code>unsigned long mode</code>	• async or sync

**Returns:** 0 if successful  
<0 if failure

**Includes:** `gclib.h`  
`gcerr.h`

**Category:** basic

**Mode:** asynchronous or synchronous

**Platform and** All†

**Technology:** †See the *Global Call Technology User's Guides* for additional information.

### ■ Description

The `gc_WaitCall()` function indicates that the application is ready to receive 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.

In the asynchronous mode, after the `gc_WaitCall()` function was successfully called, the `gc_ReleaseCallEx()` function will not block the incoming notification. Therefore, it is only necessary to call a `gc_WaitCall()` function once. However, the `gc_WaitCall()` function must be called again after issuing a `gc_Close()` or a `gc_ResetLineDev()` function.

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_ReleaseCallEx()` and `gc_WaitCall()` functions, the call will be pending until the `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.

See also the appropriate *Global Call Technology User's Guide* for technology-specific information.

Parameter	Description
<b>linedev</b>	Global Call line device handle
<b>crnp</b>	points to the CRN. The <b>crnp</b> parameter must be of a global and non-temporary type. The <b>crnp</b> 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.
<b>waitcallp</b>	not used. Set to zero.
<b>timeout</b>	(used only in synchronous mode; ignored in asynchronous mode) specifies the interval (in seconds) to wait for the call. In this case, when the time-out 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 time-out is 0 and no inbound call is pending, the function returns -1 with an EGC_TIMEOUT error value.
<b>mode</b>	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution

## ■ Termination Events

None

**Note:** In asynchronous mode, an unsolicited GCEV\_OFFERED event will be received when a call is received and the call state transitions to the Offered state.

## ■ Cautions

- The application should always call a [gc\\_ReleaseCallEx\(\)](#) 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 (that is, until the GCEV\_OFFERED event is 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.
- If this function is called synchronously without waiting forever, calls may be lost unless the next **gc\_WaitCall()** is called in less than a protocol-dependent time. It is highly recommended that either **gc\_WaitCall()** be called asynchronously or if called synchronously, wait forever (that is, use a **timeout** value of INT\_MAX).
- If the **gc\_WaitCall()** function is already active and is called again, the resulting behavior depends on the call control library being used. For some call control libraries, an error will be returned, while other call control libraries will accept multiple **gc\_WaitCall()** calls.

## ■ Errors

If this function returns `<0` to indicate failure, use the `gc_ErrorInfo()` function to retrieve the reason for the error. See the “Error Handling” section in the *Global Call API Programming Guide*. All Global Call error codes are defined in the `gcerr.h` file. If the error returned is technology specific, see the technology-specific error header file(s) for the error definition (for example, `ccerr.h` or `isdnerr.h` file for the ISDN call control library).

## ■ Example

```
#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)
{
    GC_INFO      gc_error_info;    /* GlobalCall error information data */

    /* 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_ErrorInfo(&gc_error_info);
            printf("Error: gc_WaitCall() on device handle: 0x%lx, GC ErrorValue: 0x%hx - %s,\n",
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %s\n",
                pline->ldev, gc_error_info.gcValue, gc_error_info.gcMsg,
                gc_error_info.ccLibId, gc_error_info.ccLibName,
                gc_error_info.ccValue, gc_error_info.ccMsg);
            return (gc_error_info.gcValue);
        }
    }
    /*
     * GCEV_OFFERED event will indicate incoming call has arrived.
     */
    return (0);
}
```

*gc\_WaitCall()* — indicate that the application is ready to receive inbound calls



■ See Also

- [gc\\_DropCall\(\)](#)
- [gc\\_MakeCall\(\)](#)
- [gc\\_ReleaseCallEx\(\)](#)
- [gc\\_ResetLineDev\(\)](#)



This chapter provides information about the events used by the Global Call software. Topics include:

- [Event Types](#) ..... 353
- [Event Information](#) ..... 353

## 3.1 Event Types

The Global Call software uses three different types of events:

### Termination Events

These events are returned after the termination of a function call. Note that termination events apply to the asynchronous programming model only.

### Notification Events

These events are requested by the application and provide information about a function call. Notification events can be received in either asynchronous or synchronous mode.

### Unsolicited Events

These events are not requested by the application. They are triggered by, and provide information about, external events. Unsolicited events apply to both the synchronous and asynchronous programming model.

## 3.2 Event Information

Events used by the Global Call software are listed below. For each event, the following information is provided:

- The name of the event. A dagger (†) next to the event name indicates that the name is maskable and the event's default setting (enabled or disabled) is indicated in parentheses.
- The type of the event: termination, notification, or unsolicited. See [Section 3.1, “Event Types”](#), on page 353 for more information.
- The type of entity with which the event is associated: a call (CRN), line device (LDID), or Request ID.
- A short description explaining why the event is received and what the event indicates.

**Note:** For more information about masking (enabling or disabling) events, see the description of the [gc\\_SetConfigData\( \)](#) function (if supported by the technology being used) or the [gc\\_SetEvtMsk\( \)](#) function.

The following events, listed in alphabetical order, are used by the Global Call software:

**GCEV\_ACCEPT**

Termination event for [gc\\_AcceptCall\(\)](#)

Associated with a call (CRN)

Call received at remote end, but not yet answered.

**GCEV\_ACKCALL**

Termination event for [gc\\_CallAck\(\)](#)

Associated with a call (CRN)

Indicates termination of [gc\\_CallAck\(\)](#) and that the DDI string may be retrieved by using [gc\\_GetDNIS\(\)](#).

**GCEV\_ALARM**

Unsolicited event

Associated with a line device (LDID)

An alarm occurred on a trunk or a time slot. If enabled and allowed, GCEV\_BLOCKED and GCEV\_UNBLOCKED will also be generated for blocking alarms.

**GCEV\_ALERTING† (enabled by default)**

Notification event for [gc\\_MakeCall\(\)](#)

Associated with a call (CRN)

Notifies the application that the call has reached its destination but is not yet connected to the called party. The call is waiting for the destination party to answer (ringing).

**GCEV\_ANSWERED**

Termination event for [gc\\_AnswerCall\(\)](#)

Associated with a call (CRN)

Call established and enters Connected state.

**GCEV\_ATTACH**

Termination event for [gc\\_Attach\(\)](#) or [gc\\_AttachResource\(\)](#)

Associated with a line device (LDID)

Voice or media handle was successfully attached to the line device.

**GCEV\_ATTACH\_FAIL**

Termination event for [gc\\_Attach\(\)](#) or [gc\\_AttachResource\(\)](#)

Associated with a line device (LDID)

The corresponding attach function failed.

**GCEV\_BLINDTRANSFER**

Termination event for [gc\\_BlindTransfer\(\)](#)

Associated with a call (CRN)

Generated when the call transfer has been successfully initiated in an unsupervised transfer.

**GCEV\_BLOCKED† (enabled by default)**

Unsolicited event

Associated with a line device (LDID)

Indicates that the line is blocked and application cannot issue call-related function calls.

Retrieve reason for line blockage using [gc\\_ResultValue\(\)](#).

**GCEV\_CALLINFO**

Unsolicited event

Associated with a call (CRN)

Generated when an incoming information message is received, for example, in response to a [gc\\_SndMsg\(\)](#) function call from the remote side.

**GCEV\_CALLPROC**

Termination event for [gc\\_CallAck\(\)](#)

Associated with a call (CRN)

Generated when a call proceeding indication is sent out.

**GCEV\_CALLSTATUS**

Termination event for [gc\\_MakeCall\(\)](#)

Associated with a call (CRN)

Indicates that a time-out or a no answer (call control library dependent) condition was returned while the [gc\\_MakeCall\(\)](#) function is active.

**GCEV\_COMPLETETRANSFER**

Termination event for [gc\\_CompleteTransfer\(\)](#)

Associated with a call (CRN)

Indicates that the original call has been transferred to the consultation call. Transfer completed successfully.

**GCEV\_CONGESTION**

Unsolicited event

Associated with a call (CRN)

Generated when an incoming congestion message is received indicating that the remote end is not ready to accept inbound user information.

**GCEV\_CONNECTED**

Termination event for [gc\\_MakeCall\(\)](#)

Associated with a call (CRN)

Call is connected.

**GCEV\_D\_CHAN\_STATUS**

Unsolicited event

Associated with a line device (LDID)

Generated when the status of the D channel changes as a result of an event on the D channel.

**GCEV\_DETACH**

Termination event for [gc\\_Detach\(\)](#)

Associated with a line device (LDID)

Voice or media handle was successfully detached from the line device.

**GCEV\_DETACH\_FAIL**

Termination event for [gc\\_Detach\(\)](#)

Associated with a line device (LDID)

The [gc\\_Detach\(\)](#) function failed.

**GCEV\_DETECTED†** (enabled by default)

Unsolicited event

Associated with a call (CRN)

Generated when an incoming call has been received. Indicates that the call is still in progress and is waiting for more digits before it can be offered to the application.

**GCEV\_DIALING†** (disabled by default)Notification event for [gc\\_MakeCall\(\)](#)

Associated with a call (CRN)

Generated when a call is sent out and enters the Dialing call state.

**GCEV\_DIALTONE†** (enabled by default)Termination event for [gc\\_SetUpTransfer\(\)](#)

Associated with a call (CRN)

Generated when a call is put on hold in order to make a consultation call during a supervised transfer.

**GCEV\_DISCONNECTED** (see also below)

Unsolicited event

Associated with a call (CRN)

Call is disconnected by remote end.

For ISDN applications, indicates that a Disconnect, Release Complete, or Release message was received.

The application must call the [gc\\_DropCall\(\)](#) and [gc\\_ReleaseCallEx\(\)](#) functions after this event is received to set the call state to Null.

**GCEV\_DISCONNECTED** (see also above)Termination event for [gc\\_MakeCall\(\)](#)

Associated with a call (CRN) or a line device (LDID)

A request or message was rejected by the network or the function timed out. The error detected prevents further call processing on this call.

**GCEV\_DIVERTED**

Unsolicited event

Associated with a call (CRN)

Generated when a NAM with divert information is received. Indicates that an outbound call was successfully diverted to another station (DPNSS and Q.SIG only).

**GCEV\_DROPCALL**Termination event for [gc\\_DropCall\(\)](#)

Associated with a call (CRN)

Call is disconnected and enters Idle state.

**GCEV\_ERROR**

Unsolicited event

Associated with a call (CRN) or line device (LDID)

Indicates that an internal component has failed. For more information, see the “Error Handling” section in the *Global Call API Programming Guide*.

**GCEV\_EXTENSION**

Unsolicited event

Associated with a call (CRN) or line device (LDID)

Indicates unsolicited information is received from the network or remote end point.

Information about the event is contained in the [EXTENSIONEVTBLK](#) structure, which is referenced via the `extevtdatap` pointer in the [METAEVENT](#) structure associated with the GCEV\_EXTENSION event.

**GCEV\_EXTENSIONCMPLT**

Termination event for [gc\\_Extension\(\)](#)

Associated with a call (CRN) or line device (LDID)

Indicates the successful completion of the extension function. Information about the event is contained in the [EXTENSIONEVTBLK](#) structure, which is referenced via the `extevtdatap` pointer in the [METAEVENT](#) structure associated with the GCEV\_EXTENSIONCMPLT event.

**GCEV\_FACILITY**

Unsolicited event

Associated with a line device (LDID)

Generated when an incoming Facility Request message is received.

**GCEV\_FACILITY\_ACK**

Unsolicited event

Associated with a line device (LDID)

Generated when an incoming facility ACK message is received.

**GCEV\_FACILITY\_REJ**

Unsolicited event

Associated with a line device (LDID)

Generated when an incoming Facility\_REJ message is received.

**GCEV\_FATALERROR**

Unsolicited event

Associated with a line device (LDID)

Generated when a fatal error occurs. A fatal error is an error that can cause the channel to hang. For information about handling fatal errors, see the “Error Handling” section in the *Global Call API Programming Guide*.

**GCEV\_GETCONFIGDATA**

Termination event for [gc\\_GetConfigData\(\)](#)

Associated with a call (CRN) or line device (LDID)

The configuration data has been retrieved successfully. Information about the event is contained in the [GC\\_RTCM\\_EVTDATA](#) structure.

**GCEV\_GETCONFIGDATA\_FAIL**

Termination event for [gc\\_GetConfigData\(\)](#)

Associated with a call (CRN) or line device (LDID)

The [gc\\_GetConfigData\(\)](#) function failed. Information about the event is contained in the [GC\\_RTCM\\_EVTDATA](#) structure.

**GCEV\_HOLDACK** (DPNSS and Q.SIG only)

Termination event for [gc\\_HoldCall\(\)](#)

Associated with a call (CRN)

Generated when a Hold Call Completed acknowledgment message is received. Indicates that the Hold Call request was acknowledged.

**GCEV\_HOLDCALL** (see also below)

Termination event for [gc\\_HoldCall\(\)](#) (PDKRT only)

Associated with a call (CRN)

Generated when a call successfully enters the OnHold state.

**GCEV\_HOLDCALL** (see also above)

Unsolicited event (ISDN only)

Associated with a call (CRN)

Generated when a hold current call message is received.

**GCEV\_HOLDREJ** (DPNSS and Q.SIG only)

Termination event for [gc\\_HoldCall\(\)](#)

Associated with a call (CRN)

Generated when a Hold Call Rejected acknowledgment message is received. Indicates that the Hold Call request was rejected.

**GCEV\_ISDNMSG**

Unsolicited event

Associated with a call (CRN)

Generated when an incoming unrecognized ISDN message is received.

**GCEV\_L2BFFRFULL**

Unsolicited event

Associated with a call (CRN)

Reserved for future use. Generated when the incoming layer 2 access message buffer is full.

**GCEV\_L2FRAME**

Unsolicited event

Associated with a call (CRN)

Generated when an incoming data link layer 2 access message is received.

**GCEV\_L2NOBFFR**

Unsolicited event

Associated with a call (CRN)

Generated when no free space (buffer) is available for an incoming layer 2 access message.

**GCEV\_LISTEN**

Termination event for [gc\\_Listen\(\)](#)

Associated with a line device (LDID)

Indicates that the time slot has been routed successfully.

**GCEV\_MEDIA\_ACCEPT**

Notification event for [gc\\_AnswerCall\(\)](#), [gc\\_AttachResource\(\)](#), [gc\\_MakeCall\(\)](#)

Associated with a call (CRN)

Generated when media has been established with the remote side and streaming may begin.

**GCEV\_MEDIA\_REJECT**

Notification event for [gc\\_AttachResource\(\)](#) or [gc\\_MakeCall\(\)](#)

Associated with a call (CRN)

Generated when an attempt to establish media with the remote side failed.

**GCEV\_MEDIA\_REQ**

Unsolicited event

Associated with a call (CRN)

Indicates that the remote side has proposed the establishment of media.

**GCEV\_MEDIADETECTED**

Notification event for [gc\\_MakeCall\(\)](#)

Associated with a call (CRN)

Indicates that the connected media type (the call progress information) has been identified, and the [gc\\_GetCallInfo\(\)](#) function can be used to retrieve information about the connected media type.

**Note:** This event is received only if enabled by the [gc\\_SetParm\(\)](#) function with GCPR\_MEDIADETECT.

**GCEV\_MOREDIGITS**

Termination event for [gc\\_CallAck\(\)](#)

Associated with a call (CRN)

Generated when more digits are requested. The event indicates whether:

- the requested number of digits were available
- additional digits are present
- no digits were available

**GCEV\_MOREINFO**

Termination event for [gc\\_CallAck\(\)](#)

Associated with a call (CRN)

Generated when more digits are requested. The event indicates whether:

- the requested number of digits were available
- additional digits are present
- no digits were available

**GCEV\_NOFACILITYBUF**

Notification event for [gc\\_GetCallInfo\(\)](#)

Associated with a call (CRN)

Generated when a combination of multiple IEs are received in the same message and the IEs exceed the storage capacity of the library.

**GCEV\_NOTIFY**

Unsolicited event

Associated with a call (CRN)

Generated when an incoming notify message is received.

**GCEV\_NOUSRINFOBUF† (enabled by default)**

Unsolicited event

Associated with a call (CRN)

Indicates that the incoming UUI is discarded. An incoming UUI is not accepted until the existing UUI is read by the application.

**GCEV\_NSI (DPNSS and Q.SIG only)**

Unsolicited event

Associated with a call (CRN)

Generated when a Network Specific Information (NSI) message is received.

**GCEV\_OFFERED**

Unsolicited event

Associated with a call (CRN)

Indicates that an inbound call arrived; call enters Offered state.

**GCEV\_OPENEX**

Termination event for [gc\\_OpenEx\( \)](#)

Associated with a line device (LDID)

Indicates that the device was opened successfully.

**GCEV\_OPENEX\_FAIL**

Termination event for [gc\\_OpenEx\( \)](#)

Associated with a line device (LDID)

Indicates that the request to open the device failed. A [gc\\_Close\( \)](#) function must still be performed on the line device handle to free resources.

**GCEV\_PROCEEDING† (enabled by default)**

Notification event for [gc\\_MakeCall\( \)](#)

Associated with a call (CRN)

Generated when the call is sent out and enters the proceeding state. The necessary information for the call is completed. The call is proceeding.

**GCEV\_PROGRESSING† (enabled by default)**

Notification event for [gc\\_MakeCall\( \)](#)

Associated with a call (CRN)

Generated when an incoming progress message is received.

**GCEV\_RELEASECALL**

Termination event for [gc\\_ReleaseCallEx\( \)](#)

Associated with a call (CRN)

Indicates that the call has been released.



**GCEV\_RELEASECALL\_FAIL**

Termination event for [gc\\_ReleaseCallEx\(\)](#)

Associated with a call (CRN)

Indicates that the release call action failed.

**GCEV\_REQANI**

Termination event for [gc\\_ReqANI\(\)](#)

Associated with a call (CRN)

Generated when ANI information is received from the network. Applies to the AT&T ANI-on-Demand feature only.

**GCEV\_REQMOREINFO† (enabled by default)**

Notification event for [gc\\_MakeCall\(\)](#)

Associated with a call (CRN)

Generated when the remote side requests more digits, which typically occurs after the call enters the Dialing state.

**GCEV\_RESETLINEDEV**

Termination event for [gc\\_ResetLineDev\(\)](#)

Associated with a line device (LDID)

Generated when the [gc\\_ResetLineDev\(\)](#) function completes.

**GCEV\_RESTARTFAIL**

Termination event for [gc\\_ResetLineDev\(\)](#)

Associated with a line device (LDID)

Generated when the [gc\\_ResetLineDev\(\)](#) function fails on ISDN.

**GCEV\_RETRIEVEACK (DPNSS and Q.SIG only)**

Termination event for [gc\\_RetrieveCall\(\)](#)

Associated with a call (CRN)

Generated when a Retrieve Call Completed acknowledgment is received. Indicates that the call on hold has been reconnected.

**GCEV\_RETRIEVECALL (see also below)**

Termination event for [gc\\_RetrieveCall\(\)](#) (PDKRT only)

Associated with a call (CRN)

Indicates that a call on hold was successfully retrieved.

**GCEV\_RETRIEVECALL (see also above)**

Unsolicited event (ISDN only)

Associated with a call (CRN)

Generated when a request has been received from the remote side to return a call on hold to the active state.

**GCEV\_RETRIEVEREJ (DPNSS and Q.SIG only)**

Termination event for [gc\\_RetrieveCall\(\)](#)

Associated with a call (CRN)

Generated when a Retrieve Call Rejected acknowledgment message is received. Indicates that the held call could not be reconnected.

**GCEV\_SENDMOREINFO**

Termination event for [gc\\_SendMoreInfo\(\)](#)

Associated with a call (CRN)

Generated when the information is successfully sent.

**GCEV\_SERVICEREQ**

Unsolicited event

Associated with a call (CRN) or line device (LDID)

Generated when the network receives a service request.

**GCEV\_SERVICERESP**

Termination event for [gc\\_ReqService\(\)](#)

Associated with a call (CRN) or line device (LDID)

Generated when the network receives a response from the remote device to a service request.

**GCEV\_SERVICERESPCMPLT**

Termination event for [gc\\_RespService\(\)](#)

Associated with a call (CRN) or line device (LDID)

Indicates that a response to a service request has been successfully generated.

**GCEV\_SETBILLING**

Termination event for [gc\\_SetBilling\(\)](#)

Associated with a call (CRN)

Generated when billing information for the call is acknowledged by the network. (Applies to AT&T ANI-on-Demand feature only.)

**GCEV\_SETCHANSTATE** (see also below)

Termination event for [gc\\_SetChanState\(\)](#)

Associated with a line device (LDID)

Line device was placed in requested state.

**GCEV\_SETCHANSTATE** (see also above)

Unsolicited event (ISDN only)

Associated with a line device (LDID)

Generated when the status of the B channel changes or a Maintenance message is received from the network.

If applications are both ISDN and non-ISDN, follow these steps to determine whether or not the event is unsolicited:

1. Check the call control library form (CC, not GC) of the result value.
2. Convert the ISDN *cclib* name to the *cclibid* after calling the [gc\\_Start\(\)](#) function.
3. Compare the ISDN *cclibid* to the metaevent *cclibid*; the result value for ISDN applications is non-0 for unsolicited and 0 for termination.

**GCEV\_SETCONFIGDATA**

Termination event for [gc\\_SetConfigData\(\)](#)

Associated with a Request ID

Indicates that the configuration data has been successfully updated. Information about the event is contained in the [GC\\_RTCM\\_EVTDATA](#) structure.

**GCEV\_SETCONFIGDATA\_FAIL**

Termination event for [gc\\_SetConfigData\(\)](#)

Associated with a Request ID

Indicates that an error occurred when updating the configuration data. Information about the event is contained in the [GC\\_RTCM\\_EVTDATA](#) structure.

**GCEV\_SETUP\_ACK†** (disabled by default)

Notification event for [gc\\_MakeCall\(\)](#)

Associated with a call (CRN)

Generated when an incoming setup ACK message is received.

**GCEV\_SETUPTRANSFER**

Termination event for [gc\\_SetUpTransfer\(\)](#)

Associated with a call (CRN)

Indicates that a call reference number has been allocated for the consultation call; the original call is on hold.

**GCEV\_STOPMEDIA\_REQ**

Unsolicited event

Associated with a call (CRN)

Indicates that the remote side has torn down the media channel that was previously established and that streaming should stop.

**GCEV\_SWAPHOLD**

Termination event for [gc\\_SwapHold\(\)](#)

Associated with a call (CRN)

Generated when the call on hold (pending transfer) is switched to the active state and the call that was previously active is put on hold (pending transfer).

**GCEV\_TASKFAIL**

Termination event for [gc\\_AcceptCall\(\)](#), [gc\\_AnswerCall\(\)](#), [gc\\_BlindTransfer\(\)](#), [gc\\_CallAck\(\)](#), [gc\\_CompleteTransfer\(\)](#), [gc\\_DropCall\(\)](#), [gc\\_Extension\(\)](#), [gc\\_HoldCall\(\)](#), [gc\\_Listen\(\)](#), [gc\\_MakeCall\(\)](#), [gc\\_ReqANI\(\)](#), [gc\\_ReqMoreInfo\(\)](#), [gc\\_ReqService\(\)](#), [gc\\_RespService\(\)](#), [gc\\_RetrieveCall\(\)](#), [gc\\_SendMoreInfo\(\)](#), [gc\\_SetBilling\(\)](#), [gc\\_SetChanState\(\)](#), [gc\\_SetUpTransfer\(\)](#), [gc\\_SwapHold\(\)](#), [gc\\_UnListen\(\)](#)

Associated with a call (CRN) or line device (LDID)

For functions that cause call state transitions, the event indicates a failure occurred and the call state does not change. An unsolicited error event occurred during the execution of a function. The call state does not change. For more information, see the “Error Handling” section in the *Global Call API Programming Guide*.

**GCEV\_TRANSFERACK** (DPNSS and Q.SIG only)

Unsolicited event

Associated with a call (CRN)

Generated when a Transfer Acknowledge message is received from the network. Indicates that the network accepted a request to transfer a call.

**GCEV\_TRANSFERREJ (DPNSS and Q.SIG only)**

Unsolicited event

Associated with a call (CRN)

Generated when a Transfer Reject message is received from the network. Indicates that the network rejected a request to transfer a call.

**GCEV\_TRANSIT (DPNSS and Q.SIG only)**

Unsolicited event

Associated with a call (CRN)

Generated when messages are sent via a call transferring party to the destination party after a transfer call connection is completed.

**GCEV\_UNBLOCKED† (enabled by default)**

Unsolicited event

Associated with a line device (LDID)

Line is unblocked. Application may issue call-related commands to this line device.

**GCEV\_UNLISTEN**

Termination event for [gc\\_UnListen\(\)](#)

Associated with a line device (LDID)

Indicates that the time slot has been unrouted successfully.

**GCEV\_USRINFO**

Unsolicited event

Associated with a call (CRN)

Generated when an incoming User Information message is received, for example, in response to a [gc\\_SndMsg\(\)](#) function call in which the **msg\_type** specified is SndMsg\_UsrInformation. Indicates that a User-to-User Information (UUI) event is coming.

This chapter provides an alphabetical reference to the data structures used by the Global Call software. The following data structures are described:

• ALARM_FIELD .....	367
• ALARM_LIST .....	368
• ALARM_PARM_FIELD.....	369
• ALARM_PARM_LIST .....	370
• ALARM_SOURCE_OBJECT_FIELD.....	371
• ALARM_SOURCE_OBJECT_LIST .....	372
• CCLIB_START_STRUCT .....	373
• CT_DEVINFO.....	374
• DX_CAP .....	375
• EXTENSIONEVTBLK.....	380
• GC_CALLACK_BLK.....	381
• GC_CCLIB_STATE .....	383
• GC_CCLIB_STATUS .....	384
• GC_CCLIB_STATUSALL .....	385
• GC_CUSTOMLIB_STRUCT .....	386
• GC_IE_BLK .....	387
• GC_INFO .....	388
• GC_L2_BLK.....	389
• GC_MAKECALL_BLK.....	390
• GC_PARM.....	391
• GC_PARM_BLK.....	392
• GC_PARM_DATA.....	393
• GC_PARM_ID.....	394
• GC_RATE_U.....	395
• GC_RTCM_EVTDATA.....	396
• GC_START_STRUCT .....	397
• GCLIB_ADDRESS_BLK.....	398
• GCLIB_CALL_BLK.....	400

- [GCLIB\\_CHAN\\_BLK..... 401](#)
- [GCLIB\\_MAKECALL\\_BLK ..... 402](#)
- [METAEVENT ..... 403](#)
- [SC\\_TSINFO ..... 405](#)

## ALARM\_FIELD

```
typedef struct
{
    long        alarm_number;
    GC_PARM     alarm_data;
    GC_PARM     rfu;
} ALARM_FIELD;
```

### ■ Description

The `ALARM_FIELD` structure contains information about one particular alarm in the [ALARM\\_LIST](#) data structure.

### ■ Field Descriptions

The fields of `ALARM_FIELD` are described as follows:

`alarm_number`

the alarm ID for the alarm that the information in this data structure is being referred to

`alarm_data`

if used, the data for the alarm; the information contained in the structure depends on the configuration information specified

`rfu`

reserved for future use; must be memset to 0

## ALARM\_LIST

```
typedef struct
{
    int          n_alarms;
    ALARM_FIELD  alarm_fields[MAX_NUMBER_OF_ALARMS];
} ALARM_LIST;
```

### ■ Description

The ALARM\_LIST structure contains a list of alarms for a specified alarm source object (ASO) on a given line device.

### ■ Field Descriptions

The fields of the ALARM\_LIST are described as follows:

n\_alarms

the number of alarms in the list

alarm\_fields

an array of data structures containing information about the individual alarms



## ALARM\_PARM\_FIELD

```
typedef struct
{
    GC_PARM    alarm_parm_number;
    GC_PARM    alarm_parm_data;
    GC_PARM    rfu;
} ALARM_PARM_FIELD;
```

### ■ Description

The `ALARM_PARM_FIELD` data structure represents one entry in the [ALARM\\_PARM\\_LIST](#), and contains the data for an alarm parameter.

### ■ Field Descriptions

The fields of `ALARM_PARM_FIELD` are described as follows:

`alarm_parm_number`  
the alarm parameter

`alarm_parm_data`  
the parameter data

`rfu`  
reserved for future use; must be memset to 0

## ALARM\_PARM\_LIST

```
typedef struct
{
    int                n_parms;
    ALARM_PARM_FIELD   alarm_parm_fields[MAX_NUMBER_OF_ALARM_PARAMS];
} ALARM_PARM_LIST;
```

### ■ Description

The ALARM\_PARM\_LIST data structure represents the lists of alarm parameters that are passed between the application and GCAMS. It contains the number of alarm parameters in the list as well as all the alarm parameter fields.

### ■ Field Descriptions

The fields of ALARM\_PARM\_LIST are described as follows:

n\_parms  
the number of alarm parameters in the list

alarm\_parm\_fields  
an array of data structures containing alarm parameters. See [ALARM\\_PARM\\_FIELD](#), on page 369 for a description of that data structure.

## `ALARM_SOURCE_OBJECT_FIELD`

```
typedef struct
{
    GC_PARM    aso_data;
    int        rfu;
} ALARM_SOURCE_OBJECT_FIELD;
```

### ■ Description

The `ALARM_SOURCE_OBJECT_FIELD` data structure represents one entry in the [ALARM\\_SOURCE\\_OBJECT\\_LIST](#) that corresponds to one alarm source object.

### ■ Field Descriptions

The fields of `ALARM_SOURCE_OBJECT_FIELD` are described as follows:

`aso_data`  
the data for the alarm source object

`rfu`  
reserved for future use. Set to 0.

## ALARM\_SOURCE\_OBJECT\_LIST

```
typedef struct {  
    int  
        ALARM_SOURCE_OBJECT_FIELD    n_asos;  
} ALARM_SOURCE_OBJECT_LIST;    aso_fields[MAX_ASOS];
```

### ■ Description

The ALARM\_SOURCE\_OBJECT\_LIST structure contains information about a list of alarm source objects.

### ■ Field Descriptions

The fields of ALARM\_SOURCE\_OBJECT\_LIST are described as follows:

n\_asos  
the number of alarm source objects in the aso\_fields

aso\_fields  
the fields associated with the alarm source objects

## CCLIB\_START\_STRUCT

```
typedef struct {
    char    *cclib_name;
    void     *cclib_data;
} CCLIB_START_STRUCT, *CCLIB_START_STRUCTP;
```

### ■ Description

The CCLIB\_START\_STRUCT structure contains startup information for a call control library.

### ■ Field Descriptions

The fields of CCLIB\_START\_STRUCT are described as follows:

#### cclib\_name

name of the call control library. Valid call control library names are:

- “GC\_ALL\_LIB” – All call control libraries
- “GC\_ANAPI\_LIB” – ANAPI call control library
- “GC\_CUSTOM1\_LIB” – Custom call control library 1
- “GC\_CUSTOM2\_LIB” – Custom call control library 2
- “GC\_DM3CC\_LIB” – DM3CC call control library
- “GC\_ICAPI\_LIB” – ICAPI call control library
- “GC\_IPM\_LIB” – IP\_Media call control library
- “GC\_ISDN\_LIB” – ISDN call control library
- “GC\_PDKRT\_LIB” – PDKRT call control library
- “GC\_SS7\_LIB” – SS7 call control library

#### cclib\_data

pointer to the call control specific data structure. Call control library specific information can be passed to the library through this pointer. For custom libraries, the [GC\\_CUSTOMLIB\\_STRUCT](#) can be used for loading the call control library.

## CT\_DEVINFO

```
typedef struct ct_devinfo {  
    unsigned long    ct_prodid;  
    unsigned char    ct_devfamily;  
    unsigned char    ct_devmode;  
    unsigned char    ct_nettype;  
    unsigned char    ct_busmode;  
    unsigned char    ct_busencoding;  
    unsigned char    ct_rfu[7];  
} CT_DEVINFO;
```

### ■ Description

The CT\_DEVINFO structure contains information about a specified Global Call line device.

Valid values for each member of the structure are defined in *gclib.h*. See the *Global Call Technology User's Guides* for possible return values.

### ■ Field Descriptions

The fields of CT\_DEVINFO are described as follows:

ct_prodid	Intel® Dialogic product identification
ct_devfamily	device family
ct_devmode	device mode field
ct_nettype	type of network interface
ct_busmode	specifies the bus architecture used by the device
ct_busencoding	specifies the type of PCM encoding used by the device
ct_rfu	reserved for future use

## DX\_CAP

```
typedef struct dx_cap {
    unsigned short ca_nbrdna; /* # of rings before no answer. */
    unsigned short ca_stdely; /* Delay after dialing before analysis. */
    unsigned short ca_cnosig; /* Duration of no signal time out delay. */
    unsigned short ca_lcdly; /* Delay after dial before lc drop connect */
    unsigned short ca_lcdly1; /* Delay after lc drop con. before msg. */
    unsigned short ca_hedge; /* Edge of answer to send connect message. */
    unsigned short ca_cnosil; /* Initial continuous noise timeout delay. */
    unsigned short ca_lo1tola; /* % acceptable pos. dev of short low sig. */
    unsigned short ca_lo1tolb; /* % acceptable neg. dev of short low sig. */
    unsigned short ca_lo2tola; /* % acceptable pos. dev of long low sig. */
    unsigned short ca_lo2tolb; /* % acceptable neg. dev of long low sig. */
    unsigned short ca_hi1tola; /* % acceptable pos. dev of high signal. */
    unsigned short ca_hi1tolb; /* % acceptable neg. dev of high signal. */
    unsigned short ca_lo1bmax; /* Maximum interval for shrt low for busy. */
    unsigned short ca_lo2bmax; /* Maximum interval for long low for busy. */
    unsigned short ca_hi1bmax; /* Maximum interval for 1st high for busy */
    unsigned short ca_nsbuys; /* Num. of highs after nbrdna busy check. */
    unsigned short ca_logltch; /* Silence deglitch duration. */
    unsigned short ca_higlth; /* Non-silence deglitch duration. */
    unsigned short ca_lo1rmax; /* Max. short low dur. of double ring. */
    unsigned short ca_lo2rmin; /* Min. long low dur. of double ring. */
    unsigned short ca_intflg; /* Operator intercept mode. */
    unsigned short ca_intfltr; /* Minimum signal to qualify freq. detect. */
    unsigned short rful; /* reserved for future use */
    unsigned short rfu2; /* reserved for future use */
    unsigned short rfu3; /* reserved for future use */
    unsigned short rfu4; /* reserved for future use */
    unsigned short ca_hisiz; /* Used to determine which lowmax to use. */
    unsigned short ca_alowmax; /* Max. low before con. if high >hisize. */
    unsigned short ca_blowmax; /* Max. low before con. if high <hisize. */
    unsigned short ca_nrbeg; /* Number of rings before analysis begins. */
    unsigned short ca_hilceil; /* Maximum 2nd high dur. for a retrain. */
    unsigned short ca_lo1ceil; /* Maximum 1st low dur. for a retrain. */
    unsigned short ca_lowerfrq; /* Lower allowable frequency in hz. */
    unsigned short ca_upperfrq; /* Upper allowable frequency in hz. */
    unsigned short ca_timefrq; /* Total duration of good signal required. */
    unsigned short ca_rejctfrq; /* Allowable % of bad signal. */
    unsigned short ca_maxansr; /* Maximum duration of answer. */
    unsigned short ca_ansrdgl; /* Silence deglitching value for answer. */
    unsigned short ca_mxtimefrq; /* max time for 1st freq to remain in bounds */
    unsigned short ca_lower2frq; /* lower bound for second frequency */
    unsigned short ca_upper2frq; /* upper bound for second frequency */
    unsigned short ca_time2frq; /* min time for 2nd freq to remains in bounds */
    unsigned short ca_mxtime2frq; /* max time for 2nd freq to remain in bounds */
    unsigned short ca_lower3frq; /* lower bound for third frequency */
    unsigned short ca_upper3frq; /* upper bound for third frequency */
    unsigned short ca_time3frq; /* min time for 3rd freq to remains in bounds */
    unsigned short ca_mxtime3frq; /* max time for 3rd freq to remain in bounds */
    unsigned short ca_dtn_pres; /* Length of a valid dial tone (def=1sec) */
    unsigned short ca_dtn_npres; /* Max time to wait for dial tone (def=3sec) */
    unsigned short ca_dtn_deboff; /* The dialtone off debouncer (def=100ms) */
    unsigned short ca_pamd_failtime; /* Wait for AMD/PVD after cadence */
    /* break(default=4sec) */
    unsigned short ca_pamd_minring; /* min allowable ring duration (def=1.9sec) */
    unsigned char ca_pamd_spdval; /* Set to 2 selects quick decision (def=1) */
    unsigned char ca_pamd_qtemp; /* The Qualification template to use for PAMD */
    unsigned short ca_noanswer; /* time before no answer after first ring */
    /* (default=30sec) */
    unsigned short ca_maxintering; /* Max inter ring delay before connect (8 sec) */
} DX_CAP;
```

## ■ Description

The DX\_CAP structure contains call progress information for a specified Global Call line device. For more information about the structure, see the *Voice API Library Reference* for your operating system.

## ■ Field Descriptions

The fields of DX\_CAP are described as follows:

ca\_nbrdna  
    number of rings before no answer

ca\_stdely  
    delay after dialing before call analysis

ca\_cnosig  
    duration of no signal time-out delay

ca\_lcdly  
    delay after dialing before loop current drop connect

ca\_lcdly1  
    delay after loop current drop connect before message

ca\_hedge  
    edge of answer to send connect message

ca\_cnosil  
    initial continuous noise time-out delay

ca\_lo1tola  
    percentage of acceptable positive deviation of short low signal

ca\_lo1tolb  
    percentage of acceptable negative deviation of short low signal

ca\_lo2tola  
    percentage of acceptable positive deviation of long low signal

ca\_lo2tolb  
    percentage of acceptable negative deviation of long low signal

ca\_hi1tola  
    percentage of acceptable positive deviation of high signal

ca\_hi1tolb  
    percentage of acceptable negative deviation of high signal

ca\_lo1bmax  
    maximum interval for short low busy signal

ca\_lo2bmax  
    maximum interval for long low for busy signal

ca\_hilbmax  
    maximum interval for first high for busy signal



ca\_nsbusy  
     number of highs after nbrdna (number does not answer) busy check  
 ca\_logltch  
     silence deglitch duration  
 ca\_higlth  
     non-silence deglitch duration  
 ca\_lolrmax  
     maximum short low duration of double ring  
 ca\_lo2rmin  
     minimum long low duration of double ring  
 ca\_intflag  
     operator intercept mode  
 ca\_intfltr  
     minimum signal to qualify for frequency detection  
 rfu1  
     reserved for future use  
 rfu2  
     reserved for future use  
 rfu3  
     reserved for future use  
 rfu4  
     reserved for future use  
 ca\_hisiz  
     used to determine which low maximum to use  
 ca\_alowmax  
     maximum low before connection if high >ca\_hisiz  
 ca\_blowmax  
     maximum low before connection if high <ca\_hisiz  
 ca\_nbrbeg  
     number of rings before analysis begins  
 ca\_hilceil  
     maximum second high duration for a retrain  
 ca\_lolceil  
     maximum first low duration for a retrain  
 ca\_lowerfrq  
     lower allowable frequency in Hz  
 ca\_upperfreq  
     upper allowable frequency in Hz  
 ca\_timefreq  
     total duration of good signal required

ca\_rejectfrq  
allowable percentage of bad signal

ca\_maxansr  
maximum duration of answer

ca\_ansrdgl  
silence deglitching value for answer

ca\_mxtimefrq  
maximum time for first frequency to remain in bounds

ca\_lower2freq  
lower boundary for second frequency

ca\_upper2freq  
upper boundary for second frequency

ca\_time2frq  
minimum time for second frequency to remain in bounds

ca\_mxtime2frq  
maximum time for second frequency to remain in bounds

ca\_lower3freq  
lower bound for third frequency

ca\_upper3freq  
upper bound for third frequency

ca\_time3frq  
minimum time for third frequency to remain in bounds

ca\_mxtime3frq  
maximum time for third frequency to remain in bounds

ca\_dtn\_pres  
length of a valid dial tone (default = 1 second)

ca\_dtn\_npres  
maximum time to wait for dial tone (default = 3 seconds)

ca\_dtn\_deboff  
dial tone off debouncer (default = 100 ms)

ca\_pamd\_failtime  
time to wait for AMD/PVD after cadence (default = 4 seconds)

ca\_pamd\_minring  
minimum allowable ring duration (default = 1.9 seconds)

ca\_pamd\_spdval  
set to 2 to select quick decision (default = 1)

ca\_pamd\_qtemp  
qualification template to use for PAMD

ca\_noanswer  
time before no answer after first ring (default = 30 seconds)



ca\_maxintering

maximum inter-ring delay before connection (default = 8 seconds)

## EXTENSIONEVTBLK

```
typedef struct {  
    unsigned char    ext_id;  
    GC_PARM_BLK      parmblk;  
} EXTENSIONEVTBLK;
```

### ■ Description

The extension block structure, EXTENSIONEVTBLK, contains technology-specific information and is referenced via the extevtdatap pointer in the [METAEVENT](#) structure associated with the GCEV\_EXTENSION and GCEV\_EXTENSIONCMPLT events.

### ■ Field Descriptions

The fields of EXTENSIONEVTBLK are described as follows:

ext\_id

identifier of called extension function

parmblk

[GC\\_PARM\\_BLK](#) that provides technology-specific information. See the appropriate *Global Call Technology User's Guide* for more information.

## GC\_CALLACK\_BLK

```
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 proceeding with the same B chan */
            /* 0x0001 proceeding with the new B chan */
            /* 0x0002 setup ACK */
            LINEDEV linedev;
        } isdn;
        struct {
            int info_len;
            int info_type;
        } info;
        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;
```

### ■ Description

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.

### ■ Field Descriptions

The fields of `GC_CALLACK_BLK` are described as follows:

`type`

specifies the type of service requested. The `type` field identifies the type of structure inside the service union:

- `GCACK_SERVICE_INFO` – used to request more info
- `GCACK_SERVICE_PROC` – used when call is proceeding
- `GCACK_SERVICE_DNIS` – used to request DNIS (deprecated, use `GCACK_SERVICE_INFO`)
- `GCACK_SERVICE_ISDN` – used when ISDN call is proceeding

See Table 17 for descriptions of the fields in each type of structure.

**Note:** The `GCACK_SERVICE_DNIS` and `GCACK_SERVICE_ISDN` structures are deprecated in this software release. Use the `GCACK_SERVICE_INFO` or `GCACK_SERVICE_PROC` structure instead.

`rfu`

reserved for future use; must be set to 0

service

kind of service requested; related to type field. Not used for the GCACK\_SERVICE\_PROC service type.

gc\_private[4]

for internal use by Global Call

**Table 17. Service Type Data Structure Field Descriptions**

Service Type	Data Structure Field in Union	Description
GCACK_SERVICE_DNIS (deprecated)	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. See the <i>Global Call ISDN Technology User's Guide</i> for technology-specific information.
GCACK_SERVICE_INFO	info	structure containing type and length of information requested
	info.info_type	type of information requested. The possible values are: <ul style="list-style-type: none"> <li>• DESTINATION_ADDRESS – request more DNIS</li> <li>• ORIGINATION_ADDRESS – request more ANI</li> </ul>
	info.info_len	length of the information requested (typically number of digits)
GCACK_SERVICE_ISDN (deprecated)	isdn	structure containing information for ISDN procedures supported by this function. See the <i>Global Call ISDN Technology User's Guide</i> for more details.
	isdn.acceptance	indicates type of message to be sent to network. Valid values are: <ul style="list-style-type: none"> <li>• CALL_PROCEEDING – send Proceeding message</li> <li>• CALL_SETUP_ACK – send Setup Acknowledge message</li> </ul>
	isdn.linedev	the new Global Call line device to be used for the call. If set to 0, the channel requested by the network will be used.

## GC\_CCLIB\_STATE

```
typedef struct {
    char    name[GC_MAX_CCLIBNAME_LEN];    /* name of the library. */
    int     state;                          /* state of the library */
} GC_CCLIB_STATE, *GC_CCLIB_STATEP;
```

### ■ Description

The GC\_CCLIB\_STATE structure contains the status of any call control library.

### ■ Field Descriptions

The fields of GC\_CCLIB\_STATE are described as follows:

#### name

specifies the name of the call control library and is filled in by the Global Call library. Valid call control library names are:

- “GC\_ALL\_LIB” – All call control libraries
- “GC\_ANAPI\_LIB” – ANAPI call control library
- “GC\_CUSTOM1\_LIB” – Custom call control library 1
- “GC\_CUSTOM2\_LIB” – Custom call control library 2
- “GC\_DM3CC\_LIB” – DM3CC call control library
- “GC\_ICAPI\_LIB” – ICAPI call control library
- “GC\_IPM\_LIB” – IP\_Media call control library
- “GC\_ISDN\_LIB” – ISDN call control library
- “GC\_PDKRT\_LIB” – PDKRT call control library
- “GC\_SS7\_LIB” – SS7 call control library

#### state

specifies the state of the call control library and is filled in by Global Call library. The possible states are:

- GC\_CCLIB\_CONFIGURED – library is configured; no attempt has been made to start the library
- GC\_CCLIB\_AVAILABLE – library is available; started successfully
- GC\_CCLIB\_FAILED – library failed to start

## GC\_CCLIB\_STATUS

```
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;
```

### ■ Description

The GC\_CCLIB\_STATUS structure contains the states of the individual call control library. The Global Call library is not a call control library and is therefore not included.

### ■ Field Descriptions

The fields of GC\_CCLIB\_STATUS are described as follows:

**num\_avllibraries**

the number of available call control libraries in the avllibraries field

**num\_configuredlibraries**

the number of configured call control libraries in the avllibraries field

**num\_failedlibraries**

the number of failed (did not start) call control libraries in the avllibraries field

**num\_stublibraries**

obsolete, not used

**avllibraries**

returns the name(s) of the available libraries in a string terminated with a NULL; for example, if there are two available call control libraries and they are the PDKRT and ISDN libraries, then:

- avllibraries[0] = "GC\_PDKRT\_LIB"
- avllibraries[1] = "GC\_ISDN\_LIB"

Each name is a null-terminated string.

**configuredlibraries**

an array of name(s) of the configured libraries. Each name is a null-terminated string in the same format as avllibraries

**failedlibraries**

an array of name(s) of the failed libraries. Each name is a null-terminated string in the same format as avllibraries

**stublibraries**

obsolete, not used



## GC\_CCLIB\_STATUSALL

```
typedef struct {  
    GC_CCLIB_STATE    cclib_status[MAX_CCLIBS];  
} GC_CCLIB_STATUSALL, *GC_CCLIB_STATUSALLP;
```

### ■ Description

The GC\_CCLIB\_STATUSALL structure contains the status of all call control libraries. The Global Call library is not a call control library and is therefore not included.

### ■ Field Descriptions

The fields of GC\_CCLIB\_STATUSALL are described as follows:

cclib\_status

an array of type [GC\\_CCLIB\\_STATE](#) where each element in the array contains the status of one call control library. See [GC\\_CCLIB\\_STATE](#), on page 383 for more information.

## GC\_CUSTOMLIB\_STRUCT

```
typedef struct {  
    char    *file_name;  
    char    *start_proc_name;  
} GC_CUSTOMLIB_STRUCT, *GC_CUSTOMLIB_STRUCTP;
```

### ■ Description

Third-party Global Call compatible call control libraries can be used as custom libraries. If the name in the [CCLIB\\_START\\_STRUCT](#) is either GC\_CUSTOM1\_LIB or GC\_CUSTOM2\_LIB, then the cclib\_data pointer is cast as a pointer to the GC\_CUSTOMLIB\_STRUCT structure.

### ■ Field Descriptions

The fields of GC\_CUSTOMLIB\_STRUCT are described as follows:

file\_name

file name, including the extension, of the custom library to be loaded. The name must not be greater than 8 characters long (excluding the extension).

start\_proc\_name

start procedure to be called when starting the custom library. The start procedure function name must not exceed 15 characters.

## GC\_IE\_BLK

```
typedef struct {  
    GCLIB_IE_BLK    *gclicb;  
    void            *cclicb;  
} GC_IE_BLK, *GC_IE_BLKP;
```

### ■ Description

The GC\_IE\_BLK structure is used to send an Information Element (IE) block in some technologies, for example ISDN. See the appropriate *Global Call Technology User's Guide* for technology-specific information.

### ■ Field Descriptions

The fields of GC\_IE\_BLK are described as follows:

gclicb

pointer must be set to NULL

cclicb

a pointer to IE information that is specific to the call control library (technology) being used; see the appropriate *Global Call Technology User's Guide* for technology-specific information

## GC\_INFO

```
typedef struct {  
    int     gcValue;  
    char    *gcMsg;  
    int     ccLibId;  
    char    *ccLibName;  
    long    ccValue;  
    char    *ccMsg;  
    char    *additionalInfo;  
} GC_INFO;
```

### ■ Description

The GC\_INFO structure contains error or result information for the application.

### ■ Field Descriptions

The fields of GC\_INFO are described as follows:

gcValue

Global Call error or result value

\*gcMsg

a pointer to a Global Call message associated with this error or result value

ccLibId

the ID of the call control library associated with the error or result value

\*ccLibName

a pointer to the name of the call control library associated with the error or result value

ccValue

CCLib error or result value

\*ccMsg

pointer to the CCLib message associated with this error or result value

\*additionalInfo

pointer to additional information associated with this error or result value. The string is null terminated. This additional information is optional and may be used as a diagnostic aid.

## GC\_L2\_BLK

```
typedef struct {  
    GCLIB_L2_BLK    *gclib;  
    void            *cclib;  
} GC_L2_BLK, *GC_L2_BLK_P;
```

### ■ Description

The GC\_L2\_BLK structure is used to send and receive layer 2 information to an ISDN interface. See the appropriate *Global Call Technology User's Guide* for technology-specific information; for example, information for using the cclib field.

### ■ Field Descriptions

The fields of GC\_L2\_BLK are described as follows:

gclib

pointer must be set to NULL

cclib

a pointer to L2 information that is specific to the call control library (technology) being used; see the appropriate *Global Call Technology User's Guide* for technology-specific information

## GC\_MAKECALL\_BLK

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

### ■ Description

The GC\_MAKECALL\_BLK structure contains information used by the `gc_MakeCall()` function when setting up a call.

**Note:** 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.

### ■ Field Descriptions

The fields of GC\_MAKECALL\_BLK are described as follows:

`gclib`

a pointer to a [GCLIB\\_MAKECALL\\_BLK](#) structure that contains information that is common across technologies

`cclib`

a pointer to call control library information

## GC\_PARM

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

### ■ Description

The GC\_PARM is a union of data types. The union definition is shown above.

## GC\_PARM\_BLK

```
typedef struct {  
    unsigned short    parm_data_size;  
    unsigned char     parm_data_buf[1];  
}GC_PARM_BLK, *GC_PARM_BLK_P;
```

### ■ Description

The GC\_PARM\_BLK structure contains parameter data. To retrieve or update parameter data, the parameter data must be in the format of the GC\_PARM\_DATA data structure. See [Table 21, “Possible Set ID, Parm ID Pairs used in GCLIB\\_MAKECALL\\_BLK Structure”](#), on page 421 or the appropriate *Global Call Technology User’s Guide* to find the set IDs and parameter IDs to be used in the GC\_PARM\_DATA structure. See [Figure 2, “GC\\_PARM\\_BLK Memory Diagram”](#), on page 419 for an illustration of the memory block format.

**Note:** Memory allocation and deallocation of a GC\_PARM\_BLK data block is done by the Global Call utility functions (gc\_util\_\*\*\*). See [Section 1.12, “GC\\_PARM\\_BLK Utility Functions”](#), on page 18 for more information.

### ■ Field Descriptions

The fields of GC\_PARM\_BLK are described as follows:

parm\_data\_size

the size of the parm\_data\_buf buffer in bytes

parm\_data\_buf[1]

the first byte of the first element in an array of GC\_PARM\_DATA structures. The memory for this buffer is allocated dynamically by the utility functions.



## GC\_PARM\_DATA

```
typedef struct
{
    unsigned short    set_ID;
    unsigned short    parm_ID;
    unsigned char     value_size;
    unsigned char     value_buf[1];
}GC_PARM_DATA, *GC_PARM_DATA_P;
```

### ■ Description

The GC\_PARM\_DATA structure contains parameter data. Consult the appropriate *Global Call Technology User's Guide* for additional set ID(s) and parameter ID(s).

**Note:** The set ID and the parm ID as a pair identify the parameter.

See [Figure 2, “GC\\_PARM\\_BLK Memory Diagram”](#), on page 419 for an illustration of the memory block format. See [Table 21, “Possible Set ID, Parm ID Pairs used in GCLIB\\_MAKECALL\\_BLK Structure”](#), on page 421 for possible values for the fields in the GC\_PARM\_BLK structure. Additional set IDs, parameter IDs, and parameter values can be specified by the call control library. See the appropriate *Global Call Technology User's Guide* for additional set IDs and parm IDs.

### ■ Field Descriptions

The fields of GC\_PARM\_DATA are described as follows:

set\_id

the set ID of the parameter

parm\_id

the parameter ID of the parameter

value\_size

the size of the value\_buf buffer in bytes. The size is determined by the Global Call utility functions.

value\_buf[1]

the first byte of the first element in an array of parm value buffers. The memory for this buffer is allocated dynamically by the utility functions.

## GC\_PARM\_ID

```
typedef struct
{
    unsigned short  set_ID;           /* Set ID */
    unsigned short  parm_ID;          /* Parm ID */
    unsigned char   value_type;        /* Value type, its defines listed in gccfgparm.h */
    unsigned char   update_perm;       /* Update permission */
} GC_PARM_ID, *GC_PARM_IDP;
```

### ■ Description

The GC\_PARM\_ID data structure contains configuration information returned by a [gc\\_QueryConfigData\(\)](#) query.

### ■ Field Descriptions

The fields of GC\_PARM\_ID are described as follows:

`set_id`  
the set ID of the parameter

`parm_id`  
the parameter ID of the parameter

`value_type`  
the data type of the value for this parameter (for example, int, long, etc.)

`update_perm`  
the update permission. Possible values are:

- GC\_R\_O – retrieve only
- GC\_W\_I – update immediately
- GC\_W\_N – update only while call is in the Null state
- GC\_W\_X – not available

## GC\_RATE\_U

```
typedef union {  
    struct {  
        long cents;  
    } ATT, *ATT_PTR;  
} GC_RATE_U, *GC_RATE_U_PTR;
```

### ■ Description

The GC\_RATE\_U structure is used to set billing rates for AT&T's Vari-A-Bill service.

### ■ Field Descriptions

The fields of GC\_RATE\_U are described as follows:

cents  
    defines cents as the unit for charge information

## GC\_RTCM\_EVTDATA

```
typedef struct{
    long        request_ID;
    int         gc_result;
    int         cclib_result;
    int         cclib_ID;
    char        additional_msg[MAX_ADDITIONAL_MSG];
    GC_PARM_BLK retrieved_parmblkp;
}GC_RTCM_EVTDATA, *GC_RTCM_EVTDATAP;
```

### ■ Description

The GC\_RTCM\_EVTDATA structure contains information returned via Run Time Configuration Management (RTCM) events.

The evtdatap field in the [METAEVENT](#) data structure points to this structure.

### ■ Field Descriptions

The fields of GC\_RTCM\_EVTDATA are described as follows:

request\_ID

RTCM request ID, identifies the request that triggered the event

gc\_result

Global Call result value for this event

cclib\_result

CCLib result value for this event

cclib\_ID

the ID of the CCLib associated with the CCLib result value

additional\_msg

optional additional information associated with the event. The additional information may be used as a diagnostic aid.

retrieved\_parmblkp

data associated with the event. Applies only to GCEV\_GETCONFIGDATA; field will be NULL for all other events.

## GC\_START\_STRUCT

```
typedef struct {  
    int                num_cclibs;  
    CCLIB_START_STRUCT *cclib_list;  
} GC_START_STRUCT, *GC_START_STRUCTP;
```

### ■ Description

The GC\_START\_STRUCT structure allows an application to specify which call control libraries are to be started and, optionally, to provide startup information to one or more call control libraries.

**Note:** See the appropriate *Global Call Technology User's Guide* to determine whether this structure is supported for your call control library.

### ■ Field Descriptions

The fields of GC\_START\_STRUCT are described as follows:

#### num\_cclibs

specifies the number of libraries in cclib\_list. If GC\_ALL\_LIB is specified, then all the libraries supported by Global Call are started. Otherwise, if only some libraries are to be started, the libraries must be specified in the cclib\_list field as described below.

#### cclib\_list

pointer to an array of start structures, where each structure corresponds to a library to be started. This field is ignored if num\_cclibs is set to GC\_ALL\_LIB.

## GCLIB\_ADDRESS\_BLK

```
typedef struct {
    char          address[MAX_ADDRESS_LEN];
    unsigned char address_type;
    unsigned char address_plan;
    char          sub_address[MAX_ADDRESS_LEN];
    unsigned char sub_address_type;
    unsigned char sub_address_plan;
} GCLIB_ADDRESS_BLK;
```

### ■ Description

The GCLIB\_ADDRESS\_BLK structure contains called party or calling party address information.

### ■ Field Descriptions

The fields of GCLIB\_ADDRESS\_BLK are described as follows:

#### address

specifies the address. The format is technology-specific; see the appropriate *Global Call Technology User's Guide*. Maximum size of the address is MAX\_ADDRESS\_LEN.

#### address\_type

specifies the address type. Possible values are:

- GCADDRTYPE\_TRANSPARENT – number type is transparent
- GCADDRTYPE\_NAT – national number
- GCADDRTYPE\_INTL – international number
- GCADDRTYPE\_LOC – local number
- GCADDRTYPE\_IP – Internet Protocol address
- GCADDRTYPE\_URL – URL address
- GCADDRTYPE\_DOMAIN – domain address
- GCADDRTYPE\_EMAIL – e-mail address

#### address\_plan

specifies the numbering plan for the address. This is technology-specific. Possible values are:

- GCADDRPLAN\_UNKNOWN – unknown numbering plan
- GCADDRPLAN\_ISDN – ISDN/Telephony numbering plan - E.163/E.164
- GCADDRPLAN\_TELEPHONY – telephony numbering plan - E.164
- GCADDRPLAN\_PRIVATE – private number plan

Other possible values will be provided by the technology call control library being used.

#### sub\_address

specifies the destination sub-address, typically digits. The format is technology-specific. See the appropriate *Global Call Technology User's Guide* for more information. Maximum size of the address is MAX\_ADDRESS\_LEN.

#### sub\_address\_type

specifies the destination sub-address type. Possible values are:

- GCSUBADDR\_UNKNOWN – unknown type
- GCSUBADDR\_OSI – NSAP - X.213/ISO 8348 AD2
- GCSUBADDR\_USER – user specified



- GCSUBADDR\_IA5 – IA5 digit format

Other possible values will be provided by the technology call control library being used.

sub\_address\_plan

specifies the numbering plan for the sub-address. This is technology-specific. Possible values are:

- GC\_UNKNOWN – unknown plan

Other possible values will be provided by the technology call control library being used.

## GCLIB\_CALL\_BLK

```
typedef struct {  
    unsigned char    category;  
    unsigned char    address_info;  
} GCLIB_CALL_BLK;
```

### ■ Description

The GCLIB\_CALL\_BLK structure contains call information.

### ■ Field Descriptions

The fields of GCLIB\_CALL\_BLK are described as follows:

**category**

specifies the category of the call. Possible values are:

- GCCAT\_SUB\_NOPRIOR – subscriber without priority
- GCCAT\_SUB\_PRIOR – subscriber with priority
- GCCAT\_MAINT\_EQUIP – maintenance equipment
- GCCAT\_COIN\_BOX – coinbox or subscriber with charge metering
- GCCAT\_OPERATOR – operator
- GCCAT\_DATA – data transmission
- GCCAT\_CPTP – C.P.T.P.
- GCCAT\_SPECIAL – special line
- GCCAT\_MOBILE – mobile users
- GCCAT\_VPN – virtual private network line

**address\_info**

indicates if address is partial (overlap mode) or complete (end of dialing). Possible values are:

- GCADDRINFO\_ENBLOC – address is complete
- GCADDRINFO\_OVERLAP – address is not complete



## GCLIB\_CHAN\_BLK

```
typedef struct {  
    unsigned char    medium_id;  
    unsigned char    medium_sel;  
} GCLIB_CHAN_BLK;
```

### ■ Description

The GCLIB\_CHAN\_BLK structure contains channel information.

### ■ Field Descriptions

The fields of GCLIB\_CHAN\_BLK are described as follows:

**medium\_id**

specifies the time slot or port to be connected.

**medium\_sel**

specifies if the **medium\_id** is preferred (if time slot/port is not available, use another one) or exclusive (no other time slot/port can be used). Possible values are:

- GCMEDSEL\_MEDIUM\_PREF – preferred
- GCMEDSEL\_MEDIUM\_EXCL – exclusive

See the appropriate *Global Call Technology User's Guide* for more information.

## GCLIB\_MAKECALL\_BLK

```
typedef struct {  
    GCLIB_ADDRESS_BLK    destination;  
    GCLIB_ADDRESS_BLK    origination;  
    GCLIB_CHAN_BLK       chan_info;  
    GCLIB_CALL_BLK       call_info;  
    GC_PARM_BLK          ext_datap;  
} GCLIB_MAKECALL_BLK, *GCLIB_MAKECALL_BLKP;
```

### ■ Description

The GCLIB\_MAKECALL\_BLK structure supports generic call related parameters. The GCLIB\_MAKECALL\_BLK structure above shows the fields that are common across most technologies. The fields in this structure should be initialized to a value of GCMAKECALLBLK\_DEFAULT with the exception of the ext\_datap pointer which should be set to NULL. If a value is not specified for any field, a default value will be assigned.

### ■ Field Descriptions

The fields of GCLIB\_MAKECALL\_BLK are described as follows:

destination

called party information, typically digits

origination

calling party information, typically digits

chan\_info

channel-specific information

call\_info

call-specific information

ext\_datap

pointer to the [GC\\_PARM\\_BLK](#) structure for passing additional parameters. This field can be set using (set ID, parm ID) pairs as shown in [Table 21, “Possible Set ID, Parm ID Pairs used in GCLIB\\_MAKECALL\\_BLK Structure”](#), on page 421. See the appropriate *Global Call Technology User’s Guide* to determine which fields are supported for the technology.

## METAEVENT

```
typedef struct {
    long          magicno;
    unsigned long flags;
    void          *evtdatap;
    long          evtlen;
    long          evtdev;
    long          evttype;
    LINEDEV      linedev;
    CRN          crn;
    void*         extevtdatap;
    void          *usrattr;
    int           cclibid;
    int           rful;
} METAEVENT, *METAEVENTP;
```

### Description

The METAEVENT structure contains the event descriptor for a metaevent. The [Field Descriptions](#) section below describes each element used in the metaevent data structure and indicates the function that the Global Call API uses to retrieve the information stored in the associated field. This data structure eliminates the need for the application to issue the equivalent functions listed below.

### Field Descriptions

The fields of METAEVENT are described as follows:

magicno

for internal use only

Function equivalent: none

flags

The GCME\_GC\_EVENT bit is set if, and only if, the event is a Global Call event.

Function equivalent: none

evtdatap

for internal use only

Function equivalent: **sr\_getevtdatap( )**

evtlen

for internal use only

Function equivalent: **sr\_getevtlen( )**

evtdev

event device

Function equivalent: **sr\_getevtdev( )**

evttype

event type

Function equivalent: **sr\_getevttype( )**

linedev

line device for Global Call events

Function equivalent: [gc\\_GetLineDev\(\)](#)

crn

call reference number for Global Call events; if 0, no CRN for this event

Function equivalent: [gc\\_GetCRN\(\)](#)

extevtdatap

pointer to additional event data

Function equivalent: none

usrattr

user assigned attribute associated with the line device

Function equivalent: [gc\\_GetUsrAttr\(\)](#)

cclibid

identification of call control library associated with the event:

- n = cclib ID number
- -1 = unknown

Function equivalent: none

rfu1

reserved for future use

Function equivalent: none

## SC\_TSINFO

```
typedef struct {  
    unsigned long    sc_numts;  
    long             *sc_tsarrayp;  
} SC_TSINFO;
```

### ■ Description

The SC\_TSINFO structure contains CT Bus time slot information for a specified line device.

### ■ Field Descriptions

The fields of SC\_TSINFO are described as follows:

sc\_numts

specifies the number of time slots requested (usually 1)

sc\_tsarrayp

pointer to an array. When the structure is returned, the first element of the array will contain the CT Bus time slot number on which the channel transmits.



This chapter describes the error codes used by the Global Call software.

The Global Call software uses two types of error codes as follows:

Error codes (prefixed by EGC\_)

Codes that are generated when a Global Call API function returns a value less than 0. The [gc\\_ErrorInfo\(\)](#) function can be used to retrieve the error information.

Result values (prefixed by GCRV\_)

Values that are generated when a GCEV\_TASKFAIL event is received by the application. The [gc\\_ResultInfo\(\)](#) function can be used to retrieve the error information.

Common error codes and result values are defined in the *gcerr.h* header file.

**Note:** When a Global Call API function fails, the underlying call control library may also generate an error code that contains more detail about the error. The [gc\\_ErrorInfo\(\)](#) function also provides access to this error information. The error codes for underlying call control libraries are defined in separate technology-specific header files.

Error codes (prefixed by EGC\_) and result values (prefixed by GCRV\_) are listed alphabetically and described below. The error codes include the Real Time Configuration Management (RTCM) error codes. See the *Global Call API Programming Guide* for more information about error handling.

EGC\_AGGETXMITSLLOT

[ag\\_getxmitslot\(\)](#) function failed

EGC\_AGLISTEN

Analog device listen failed

EGC\_AGUNLISTEN

Analog device unlisten failed

EGC\_ALARM

Function interrupted by alarm

EGC\_ALARMDBINIT

Alarm database failed to initialize

EGC\_ATTACHED

Specified resource already attached

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_CONGESTION	Congestion on the line
EGC_CPERROR	Special Information Tone (SIT) detection error
EGC_DEVICE	Invalid device handle
EGC_DIALTONE	No dial tone detected
EGC_DRIVER	Driver error
EGC_DTGETXMITSLLOT	<b>dt_getxmitslot( )</b> function failed
EGC_DTLISTEN	Network interface (dti) device listen failed
EGC_DTOPEN	<b>dt_open( )</b> function failed
EGC_DTUNLISTEN	Network interface (dti) device unlisten failed
EGC_DUPENTRY	Duplicate entry inserted into Global Call database
EGC_DXGETXMITSLLOT	<b>dx_getxmitslot( )</b> function failed
EGC_DXLISTEN	Voice device listen failed
EGC_DXOPEN	<b>dx_open( )</b> function failed
EGC_DXUNLISTEN	Voice device unlisten failed
EGC_FATALERROR_ACTIVE	Recovery from a fatal error is in progress
EGC_FATALERROR_OCCURRED	Fatal error occurred
EGC_FILEOPEN	Error opening file
EGC_FILEREAD	Error reading file



EGC_FILEWRITE	Error writing file
EGC_FUNC_NOT_DEFINED	Protocol function not defined
EGC_GC_STARTED	Global Call library is already started
EGC_GCDBERR	Global Call database error
EGC_GCNOTSTARTED	Global Call not started
EGC_GETXMITSLLOT	<a href="#">gc_GetXmitSlot()</a> function failed
EGC_GLARE	<a href="#">gc_MakeCall()</a> function failed due to a glare condition
EGC_ILLSTATE	Function not supported in the current state
EGC_INTERR	Internal Global Call error
EGC_INVCRN	Invalid call reference number
EGC_INVATABUFFSIZE	(RTCM) Invalid parm data buffer size
EGC_INVDEVNAME	Invalid device name
EGC_INVLINDEV	Invalid line device passed
EGC_INVMETAEVENT	Invalid metaevent
EGC_INVPARAM	Invalid parameter (argument)
EGC_INVPARAM_CCLIB	(RTCM) Invalid parm to CCLib
EGC_INVPARAM_FIRMWARE	(RTCM) Invalid parm to firmware
EGC_INVPARAM_GCLIB	(RTCM) Invalid parm to GCLib
EGC_INVPARAM_PROTOCOL	(RTCM) Invalid parm to protocol
EGC_INVPARAM_TARGET	(RTCM) Invalid parm to target object

EGC_INVPARMBLK	(RTCM) Invalid GC_PARM_BLK
EGC_INVPARMID	(RTCM) Invalid parm ID
EGC_INVPROTOCOL	Invalid protocol name
EGC_INVQUERYID	(RTCM) Invalid query ID
EGC_INVSETID	(RTCM) Invalid set ID
EGC_INVSTATE	Invalid state
EGC_INVTARGETID	(RTCM) Invalid target object ID
EGC_INVTARGETTYPE	(RTCM) Invalid target object type
EGC_INVUPDATEFLAG	(RTCM) Invalid update condition flag
EGC_LINERELATED	Error is related to line device
EGC_LISTEN	<a href="#">gc_Listen()</a> function failed
EGC_MAXDEVICES	Exceeded maximum devices limit
EGC_NAMENOTFOUND	Trunk device name not found
EGC_NDEVICE	Too many devices opened
EGC_NEXT_PARM_ERR	(RTCM) Can not get next parm data
EGC_NO_MORE_AVL	No more information available
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

EGC_NON_RECOVERABLE_FATALERROR	A non-recoverable fatal error occurred. The application must be shut down and restarted.
EGC_NORB	No ringback detected
EGC_NOT_INSERVICE	Called number is not in-service
EGC_NOVOICE	Call needs voice resource, use <a href="#">gc_AttachResource()</a> function
EGC_NPROTOCOL	Too many protocols opened
EGC_OPENH	(same as EGC_DXOPEN)
EGC_PARM_DATATYPE_ERR	(RTCM) Parm data type error
EGC_PARM_UPDATEPERM_ERR	(RTCM) Parm update not allowed
EGC_PARM_VALUE_ERR	(RTCM) Parm value error
EGC_PARM_VALUESIZE_ERR	(RTCM) Parm value buffer size error
EGC_PFILE	Error opening parameter file
EGC_PROTOCOL	Protocol error
EGC_PUTEVT	Error queuing event
EGC_QUERYEDATA_ERR	(RTCM) Query source data error
EGC_RECOVERABLE_FATALERROR	Recoverable fatal error occurred. Channel must be closed and then reopened.
EGC_REJECT	Call is rejected
EGC_RESETABLE_FATALERROR	Resetable fatal error occurred. Line device was automatically reset.
EGC_ROUTEFAIL	Routing failed
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_UNALLOCATED	Number is unallocated
EGC_UNLISTEN	<a href="#">gc_UnListen()</a> function failed
EGC_UNSUPPORTED	Function not supported by this technology
EGC_USER	Function interrupted by user
EGC_USRATTRNOTSET	User attribute for this line device not set
EGC_VOICE	No voice resource attached
EGC_VOXERR	Error from voice software
EGC_XMITALRM	Send alarm failed
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\_CONGESTION  
Congestion

GCRV\_CPERROR  
Special Information Tone (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

GCRV\_FATALERROR\_ACTIVE  
Recovery from fatal error in progress

GCRV\_FATALERROR\_OCCURRED  
Fatal error occurred

GCRV\_FERR  
Frame bit error

GCRV\_FERROK  
Frame bit error recovered

GCRV\_FSERR  
Frame sync error

GCRV\_FSERROK  
Frame sync error recovered

GCRV\_GLARE  
Glare condition occurred

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_NODYNMEM	Fatal error occurred due to no dynamic memory
GCRV_NONRECOVERABLE_FATALERROR	Non-recoverable fatal error occurred; application must be shut down and restarted
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 <a href="#">gc_AttachResource( )</a> 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
GCRV_RDMAOK	Received distant multi-frame alarm recovered
GCRV_RECOVERABLE_FATALERROR	Recoverable fatal error occurred

GCRV_RED	Received a red alarm condition
GCRV_REDOK	Received a red alarm condition recovered
GCRV_REJECT	Call rejected
GCRV_RESETTABLE_FATALERROR	Resettable fatal error occurred. The protocol was automatically reset.
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
GCRV_UNALLOCATED	Number not allocated





# Supplementary Reference Information

## 6

This chapter provides reference information about the following subjects:

- Alarm Source Object IDs ..... 417
- Target Objects ..... 417

### 6.1 Alarm Source Object IDs

This section contains information pertaining to the Global Call Alarm Management System (GCAMS).

Table 18 lists the Alarm Source Object (ASO) IDs that are used by Global Call. Only those ASOs that are known to Global Call are included in the table. For other ASO IDs, see the appropriate *Global Call Technology User's Guide*.

**Table 18. Alarm Source Object IDs**

Alarm Source Object ID	Description
ALARM_SOURCE_ID_SPRINGWARE_E1	Springware E-1 alarm source object ID
ALARM_SOURCE_ID_SPRINGWARE_T1	Springware T-1 alarm source object ID
ALARM_SOURCE_ID_DM3_E1	DM3 E-1 alarm source object ID
ALARM_SOURCE_ID_DM3_T1	DM3 T-1 alarm source object ID
ALARM_SOURCE_ID_NETWORK_ID	Network alarm source object ID. May be used instead of the actual network ID.

### 6.2 Target Objects

This section provides information related to the various target objects used by the Global Call API.

Table 19 shows the combinations of physical or logical entities and software module entities that can make up a target (**target\_type**).

Table 19. Supported Target Types

Software Module	Entity			
	System	Network Interface	Channel	CRN
GCLib	S	S	S	S
CCLib	S	S	S	S
Protocol	SV	SV	SV	
Firmware		SV	SV	
S = Supported SV = Supported with variances; see appropriate <i>Global Call Technology User's Guide</i>				

Table 20 shows the target types, as described in Table 19, with various target IDs to represent valid target objects.

Table 20. Target Type and Target ID Pairs

Target Type	Target ID	Description
GCTGT_GCLIB_SYSTEM†	GCGV_LIB(0)	Global Call library module target object
GCTGT_CCLIB_SYSTEM†	CCLib ID	Call control library module target object
GCTGT_PROTOCOL_SYSTEM†	Protocol ID	Protocol module target object
GCTGT_GCLIB_NETIF	Global Call Line device ID	Network Interface target object in Global Call library module
GCTGT_CCLIB_NETIF	Global Call Line device ID	Network Interface target object in call control library module
GCTGT_PROTOCOL_NETIF	Global Call Line device ID	Network Interface target object in Protocol module
GCTGT_FIRMWARE_NETIF†	Global Call Line device ID	Network Interface target object in Firmware module
GCTGT_GCLIB_CHAN	Global Call Line device ID	Channel target object in Global Call library module
GCTGT_CCLIB_CHAN	Global Call Line device ID	Channel target object in call control library module
GCTGT_PROTOCOL_CHAN	Global Call Line device ID	Channel of protocol module target object
GCTGT_FIRMWARE_CHAN	Global Call Line device ID	Channel target object in firmware module
GCTGT_GCLIB_CRN	Global Call CRN	CRN target object in Global Call library module
GCTGT_CCLIB_CRN	Global Call CRN	CRN target object in call control library module
†Can only be used by functions issued in synchronous mode.		

Figure 2 gives the memory diagram for the [GC\\_PARM\\_BLK](#) structure.

Figure 2. GC\_PARM\_BLK Memory Diagram

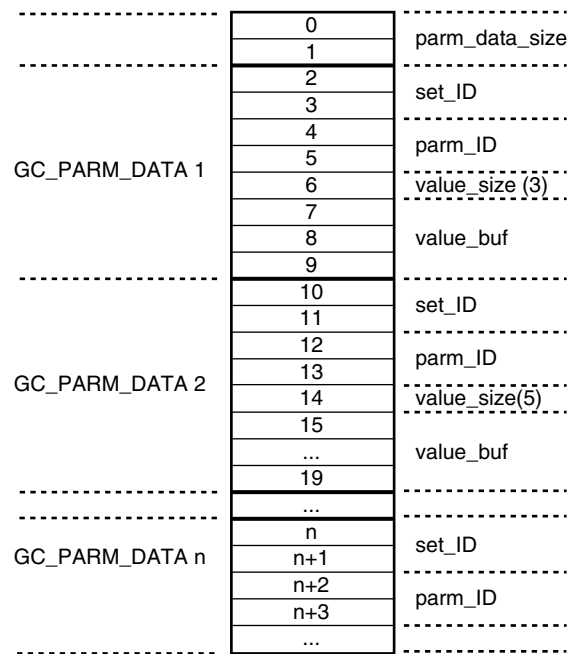


Figure 3 is an example of the GC\_PARM\_BLK structure filled with some parameters for IP technology.

Figure 3. Sample GC\_PARM\_BLK Memory Diagram

----- parm_data_size -----	16	Size in bytes of the data in the parm_dat_buf
set_ID	GCSET_CAPABILITY	Set defined by GlobalCall
parm_ID	GCPARM_TYPE	Parameter defined by GlobalCall
value_size	1	Size in bytes of the value
value_buff	GCCAPTYPE_AUDIO	
----- set_ID	GCSET_CAPABILITY	Set defined by GlobalCall
parm_ID	GCPARM_CAPABILITY	Parameter defined by GlobalCall
value_size	1	Size in bytes of the value
value_buff	GCCAP_AUDIO_g711Alaw	
----- set_ID	GCSET_CAPABILITY	Set defined by GlobalCall
parm_ID	GCPARM_RATE	Parameter defined by GlobalCall
value_size	1	Size in bytes of the value
value_buff	GCCAPRATE_64K	
----- set_ID	GCIS_SET_CHANSTATE	Set defined by ISDN library
parm_ID	GCIS_PARM_BCHANSTATE	Parameter defined by ISDN library
value_size	1	Size in bytes of the value
value_buff -----	ISDN_ACCUNET	

Table 21 shows possible (set ID, parm ID) pairs for passing additional parameters in the [GCLIB\\_MAKECALL\\_BLK](#) structure. The Value column show the possible values for the parameter IDs.

**Table 21. Possible Set ID, Parm ID Pairs used in GCLIB\_MAKECALL\_BLK Structure**

Set ID†	Parameter ID†	Value†
GCSET_DEST_ADDR	GCPARM_ADDR_DATA	technology-specific format
	GCPARM_ADDR_TYPE	GCADDRTYPE_TRANSPARENT GCADDRTYPE_NAT GCADDRTYPE_INTL GCADDRTYPE_LOC GCADDRTYPE_IP GCADDRTYPE_URL GCADDRTYPE_DOMAIN GCADDRTYPE_EMAIL
	GCPARM_ADDR_PLAN	GCADDRPLAN_UNKNOWN GCADDRPLAN_ISDN GCADDRPLAN_TELEPHONY GCADDRPLAN_PRIVATE
	GCPARM_SUBADDR_DATA	technology-specific format
	GCPARM_SUBADDR_TYPE	GCSUBADDR_UNKNOWN GCSUBADDR_OSI GCSUBADDR_USER GCSUBADDR_IA5
	GCPARM_SUBADDR_PLAN	protocol specific
GCSET_ORIG_ADDR	GCPARM_ADDR_DATA	technology-specific format
	GCPARM_ADDR_TYPE	GCADDRTYPE_TRANSPARENT GCADDRTYPE_NAT GCADDRTYPE_INTL GCADDRTYPE_LOC GCADDRTYPE_IP GCADDRTYPE_URL GCADDRTYPE_DOMAIN GCADDRTYPE_EMAIL
	GCPARM_ADDR_PLAN	GCADDRPLAN_UNKNOWN GCADDRPLAN_ISDN GCADDRPLAN_TELEPHONY GCADDRPLAN_PRIVATE
	GCPARM_SUBADDR_DATA	technology-specific format
	GCPARM_SUBADDR_TYPE	GCSUBADDR_UNKNOWN GCSUBADDR_OSI GCSUBADDR_USER GCSUBADDR_IA5
	GCPARM_SUBADDR_PLAN	protocol specific
† More set IDs, parameter IDs, and parameter values can be specified by the call control library.		

Table 21. Possible Set ID, Parm ID Pairs used in GCLIB\_MAKECALL\_BLK Structure

Set ID†	Parameter ID†	Value†
GCSET_CHAN_BLK	GCPARM_CHAN_MEDIA_ID	GC_DISABLE GC_ENABLE
	GCPARM_CHAN_MEDIA_SEL	GCMEDSEL_MEDIUM_PREF GCMEDSEL_MEDIUM_EXCL
GCSET_CALL_BLK	GCPARM_CALL_CATEGORY	GCCAT_SUB_NOPRIOR GCCAT_SUB_PRIOR GCCAT_MAINT_EQUIP GCCAT_COIN_BOX GCCAT_OPERATOR GCCAT_DATA GCCAT_CPTP GCCAT_SPECIAL GCCAT_MOBILE GCCAT_VPN
	GCPARM_CALL_ADDR_INFO	GCADDRINFO_ENBLOC GCADDRINFO_OVERLAP
† More set IDs, parameter IDs, and parameter values can be specified by the call control library.		

The parameter lists for the [gc\\_SetConfigData\(\)](#) and [gc\\_GetConfigData\(\)](#) functions are shown in Table 22 and Table 23.

Table 22. GC Parameter Entry List Maintained in GCLIB

Set ID	Parm ID	Target Object Type	Description	Data Type	Access Attribute†
GCSET_DEVICEINFO	GCPARM_DEVICENAME	GCTGT_GCLIB_NETIF GCTGT_GCLIB_CHAN	Device name	string	GC_R_O
GCSET_DEVICEINFO	GCPARM_NETWORKH	GCTGT_GCLIB_NETIF GCTGT_GCLIB_CHAN	Network device handle	int	GC_R_O
GCSET_DEVICEINFO	GCPARM_VOICENAME	GCTGT_GCLIB_CHAN	Voice device name	string	GC_R_O
GCSET_DEVICEINFO	GCPARM_VOICEH	GCTGT_GCLIB_CHAN	Voice device handle	int	GC_R_O
GCSET_CALLEVENT_MSK	GCPARM_GET_MSK	GCTGT_GCLIB_CHAN	Get call event mask	long	GC_R_O
GCSET_CALLEVENT_MSK	GCACT_SETMSK	GCTGT_GCLIB_CHAN	Set call event mask	long	GC_W_N
GCSET_CALLEVENT_MSK	GCACT_ADDMSK	GCTGT_GCLIB_CHAN	Add call event mask	long	GC_W_N
† Possible values are: GC_R_O - Retrieve only GC_W_I - Update immediately GC_W_N - Update only while call is in the Null state GC_W_X - Not available					

**Table 22. GC Parameter Entry List Maintained in GCLIB (Continued)**

Set ID	Parm ID	Target Object Type	Description	Data Type	Access Attribute†
GCSET_CALLEVENT_MSK	GCACT_SUBMSK	GCTGT_GCLIB_CHAN	Sub call event mask	long	GC_W_N
GCSET_CALLSTATE_MSK	GCPARM_GET_MSK	GCTGT_GCLIB_CHAN	Get call state mask	long	GC_R_O
GCSET_CALLSTATE_MSK	GCACT_SETMSK	GCTGT_GCLIB_CHAN	Set call state mask	long	GC_W_N
GCSET_CALLSTATE_MSK	GCACT_ADDMSK	GCTGT_GCLIB_CHAN	Add call state mask	long	GC_W_N
GCSET_CALLSTATE_MSK	GCACT_SUBMSK	GCTGT_GCLIB_CHAN	Sub call state mask	long	GC_W_N
GCSET_CRN_INDEX	GCPARM_1ST_CRN	GCTGT_GCLIB_CHAN	1st CRN	long	GC_R_O
GCSET_CRN_INDEX	GCPARM_2ND_CRN	GCTGT_GCLIB_CHAN	2nd CRN	long	GC_R_O
GCSET_DEVICEINFO	GCPARM_CALLSTATE	GCTGT_GCLIB_CRN	Call state	int	GC_R_O
GCSET_DEVICEINFO	GCPARM_BOARD_LDID	GCTGT_GCLIB_NETIF GCTGT_GCLIB_CHAN	Network interface board ID	long	GC_R_O
GCSET_PROTOCOL	GCPARM_PROTOCOL_ID	GCTGT_GCLIB_NETIF GCTGT_GCLIB_CHAN	Protocol ID	long	GC_R_O
GCSET_PROTOCOL	GCPARM_PROTOCOL_NAME	GCTGT_GCLIB_NETIF GCTGT_GCLIB_CHAN	Protocol Name	string	GC_R_O
GCSET_CCLIB_INFO	GCPARM_CCLIB_ID	GCTGT_GCLIB_NETIF GCTGT_GCLIB_CHAN GCTGT_PROTOCOL_SYSTEM	CCLib ID	long	GC_R_O
GCSET_CCLIB_INFO	GCPARM_CCLIB_NAME	GCTGT_GCLIB_NETIF GCTGT_GCLIB_CHAN GCTGT_PROTOCOL_SYSTEM	CCLib ID	string	GC_R_O
† Possible values are: GC_R_O - Retrieve only GC_W_I - Update immediately GC_W_N - Update only while call is in the Null state GC_W_X - Not available					

Table 23. Examples of Parameter Entry List Maintained in CCLIB

Set ID	Parm ID	Target Object Type	Description	Data Type	Access Attribute†
GCSET_CALLINFO	CALLINFOTYPE	GCTGT_CCLIB_CRN	Calling info type	string	GC_R_O
GCSET_CALLINFO	CATEGORY_DIGIT	GCTGT_CCLIB_CRN	Category digit	char	GC_R_O
GCSET_CALLINFO	CONNECT_TYPE	GCTGT_CCLIB_CRN	Connect type	char	GC_R_O
GCSET_PARM	GCPR_CALLINGPARTY	GCTGT_GCLIB_CHAN	Calling party	string	GC_W_I
GCSET_PARM	GCPR_LOADTONES	GCTGT_GCLIB_CHAN	Load tones	short	GC_W_I
GCSET_ORIG_ADDR	GCPARM_ADDR_DATA	GCTGT_GCLIB_CHAN	Calling number	string	GC_W_I
† Possible values are: GC_R_O - Retrieve only GC_W_I - Update immediately GC_W_N - Update only while call is in the Null state GC_W_X - Not available					

**Note:** The parameter entries shown in Table 23 are examples only. The configurable parameters depend on the CCLib implementation.



## A

- Advanced Call Model functions, list of 13
- alarm database 407
- ALARM\_FIELD data structure 367
- ALARM\_LIST data structure 368
- ALARM\_PARM\_FIELD data structure 369
- ALARM\_PARM\_LIST data structure 370
- ALARM\_SOURCE\_OBJECT\_FIELD data structure 371
- ALARM\_SOURCE\_OBJECT\_LIST data structure 372
- ANI information
  - retrieving 112
- ani\_buf buffer 112, 230
- ANI-on-Demand 229
- attaching a resource to a line device 49
- available libraries 317

## B

- B\_channel 149
  - setting information element 298
- backward compatibility
  - gc\_GetLineDev(\_) 147
- Basic Call Handling functions, list of 11
- billing\_buf buffer 114
- bitmask values, event masks 294

## C

- call control libraries
  - library identification code 65
  - opening with gc\_Start(\_) 317
- call control library 407
- call notification events 349
- call progress tone 60
- caller ID 229
- CALLNAME 116
- calls
  - causes for dropping 85
- CALLTIME 116
- CATEGORY\_DIGIT 117
- CCLIB\_START\_STRUCT data structure 373
- CONNECT\_TYPE 117

## CRN

- gc\_CRN2LineDev(\_) 79
- matching to LDID 79

## CRN (Call Reference Number)

- caution re. releasing 85, 350

- CT\_DEVINFO data structure 374

## D

- D\_channel 149

### data structures

- ALARM\_FIELD 367
- ALARM\_LIST 368
- ALARM\_PARM\_FIELD 369
- ALARM\_PARM\_LIST 370
- ALARM\_SOURCE\_OBJECT\_FIELD 371
- ALARM\_SOURCE\_OBJECT\_LIST 372
- CCLIB\_START\_STRUCT 373
- CT\_DEVINFO 374
- DX\_CAP 376
- EXTENSIONEVTBLK 380
- GC\_INFO 388
- GC\_CALLACK\_BLK 381
- GC\_CCLIB\_STATE 383
- GC\_CCLIB\_STATUS 384
- GC\_CCLIB\_STATUSALL 385
- GC\_IE\_BLK 387
- GC\_L2\_BLK 389
- GC\_MAKECALL\_BLK 390
- GC\_PARM 391
- GC\_PARM\_BLK 392
- GC\_PARM\_DATA 393
- GC\_RATE\_U 395
- GC\_RTCM\_EVTDATA 396
- GC\_START\_STRUCT 397
- GCLIB\_ADDRESS\_BLK 398
- GCLIB\_CALL\_BLK 400
- GCLIB\_CHAN\_BLK 401
- GCLIB\_MAKECALL\_BLK 402
- METAEVENT 403
- SC\_TSINFO 405

### DDI digits

- requesting and retrieving 56
- specified in dnls service structure 381

### deprecated function

- gc\_GetNetCRV(\_) 161

## deprecated functions

- gc\_GetANI(\_) 112
- gc\_Attach(\_) 48
- gc\_CCLibStatus(\_) 67
- gc\_CCLibStatusAll(\_) 69
- gc\_ErrorValue(\_) 89
- gc\_GetDNIS(\_) 138
- gc\_GetInfoElem(\_) 144
- gc\_Open(\_) 213
- gc\_ReleaseCall(\_) 226
- gc\_ResultMsg(\_) 247
- gc\_ResultValue(\_) 249
- gc\_SetInfoElem(\_) 298
- gc\_SndFrame(\_) 311
- gc\_SndMsg(\_) 141, 314
- list of 19

devicename string 215

dnis service structure 381

drop and insert configurations 60

## dropped calls

- causes 85

DX\_CAP data structure 376

# E

EGC\_TIMEOUT 350

## error codes

- definition 407
- from call control library 407

## event

- GCEV\_ACCEPT 354
- GCEV\_ACKCALL 354
- GCEV\_ALARM 354
- GCEV\_ALERTING 354
- GCEV\_ANSWERED 354
- GCEV\_ATTACH 354
- GCEV\_ATTACHFAIL 354
- GCEV\_BLOCKED 354
- GCEV\_CALLINFO 355
- GCEV\_CALLPROC 355
- GCEV\_CALLSTATUS 355
- GCEV\_COMPLETETRANSFER 355
- GCEV\_CONGESTION 355
- GCEV\_CONNECTED 355
- GCEV\_D\_CHAN\_STATUS 355
- GCEV\_DETACH 355
- GCEV\_DETACH\_FAIL 355
- GCEV\_DETECTED 356
- GCEV\_DIALING 356
- GCEV\_DIALTONE 356
- GCEV\_DISCONNECTED 356
- GCEV\_DIVERTED 356
- GCEV\_DROP CALL 356

- GCEV\_EXTENSION 357
- GCEV\_EXTENSIONCMPLT 357
- GCEV\_FACILITY 357
- GCEV\_FACILITY\_ACK 357
- GCEV\_FACILITY\_REJ 357
- GCEV\_FATALERROR 357
- GCEV\_GETCONFIGDATA 357
- GCEV\_GETCONFIGDATA\_FAIL 357
- GCEV\_HOLDACK 358
- GCEV\_HOLDCALL 358
- GCEV\_HOLDREJ 358
- GCEV\_ISDNMSG 358
- GCEV\_L2BFFRFULL 358
- GCEV\_L2FRAME 358
- GCEV\_L2NOBFFR 358
- GCEV\_MEDIA\_ACCEPT 359
- GCEV\_MEDIA\_REJECT 359
- GCEV\_MEDIA\_REQ 359
- GCEV\_MEDIADETECTED 359
- GCEV\_MOREDIGITS 359
- GCEV\_MOREINFO 359
- GCEV\_NOFACILITYBUF 359
- GCEV\_NOTIFY 360
- GCEV\_NOUSRINFOBUF 360
- GCEV\_NST 360
- GCEV\_OFFERED 360
- GCEV\_OPENEX 360
- GCEV\_OPENEX\_FAIL 360
- GCEV\_PROCEEDING 360
- GCEV\_PROGRESSING 360
- GCEV\_RELEASECALL 360
- GCEV\_RELEASECALL\_FAIL 361
- GCEV\_REQANI 361
- GCEV\_REQMOREINFO 361
- GCEV\_RESETLINEDEV 361
- GCEV\_RESTARTFAIL 361
- GCEV\_RETRIEVEACK 361
- GCEV\_RETRIEVECALL 361
- GCEV\_RETRIEVEREJ 361
- GCEV\_SENDMOREINFO 362
- GCEV\_SERVICEREQ 362
- GCEV\_SERVICERESP 362
- GCEV\_SERVICERESPCMPLT 362
- GCEV\_SETBILLING 362
- GCEV\_SETCHANSTATE 362
- GCEV\_SETCONFIGDATA 362
- GCEV\_SETCONFIGDATA\_FAIL 363
- GCEV\_SETUP\_ACK 363
- GCEV\_SETUPTRANSFER 363
- GCEV\_STOPMEDIA\_REQ 363
- GCEV\_SWAPHOLD 363
- GCEV\_TASKFAIL 363
- GCEV\_TRANSFERACK 363
- GCEV\_TRANSFERREJ 364



- GCEV\_TRANSIT 364
- GCEV\_UNBLOCKED 364
- GCEV\_USRINFO 364
- event bitmask 294
- event types
  - notification events 353
  - termination events 353
  - unsolicited events 353
- EXTENSIONEVTBLK data structure 380

## F

- failed libraries 317
- Feature Transparency and Extension functions, list of 18
- forced release 85
- FTE functions, list of 18
- function categories
  - advanced call model functions 13
  - basic functions 11
  - FTE functions 18
  - GC\_PARM\_BLK utility functions 18
  - GCAMS functions 16
  - GCSR functions 18
  - ISDN-specific functions 15
  - library information functions 12
  - optional call handling functions 12
  - RTCM functions 18
  - system controls and tools functions 13
  - voice and media functions 15
- function return value
  - GC\_SUCCESS 27
- function support by technology 20
- function syntax convention 27

## G

- gc\_AcceptCall(\_) 28
- gc\_AlarmName(\_) 31
- gc\_AlarmNumber(\_) 33
- gc\_AlarmNumberToName(\_) 35
- gc\_AlarmSourceObjectID(\_) 37
- gc\_AlarmSourceObjectIDToName(\_) 39
- gc\_AlarmSourceObjectName(\_) 41
- gc\_AlarmSourceObjectNameToID(\_) 43
- gc\_AnswerCall(\_) 45
- gc\_Attach(\_) 48
- gc\_Attach()
  - deprecated function 48
- gc\_AttachResource(\_) 49
- gc\_BlindTransfer(\_) 53
- GC\_CALL\_REJECTED 85
- gc\_CallAck(\_) 56
- GC\_CALLACK\_BLK data structure 381
- gc\_CallProgress(\_) 60
- GC\_CCLIB\_AVL 67
- GC\_CCLIB\_CONFIGURED 67
- GC\_CCLIB\_FAILED 67
- GC\_CCLIB\_STATE data structure 383
- GC\_CCLIB\_STATUS data structure 384
  - use by gc\_CCLibStatusAll(\_) 69
- GC\_CCLIB\_STATUSALL data structure 385
- gc\_CCLibIDToName(\_) 63
- gc\_CCLibNameToID(\_) 65
- gc\_CCLibStatus(\_) 67
  - deprecated function 67
- gc\_CCLibStatusAll(\_) 69
  - deprecated function 69
- gc\_CCLibStatusEx(\_) 71, 317
- GC\_CHANNEL\_UNACCEPTABLE 85
- gc\_Close(\_) 74
- gc\_CompleteTransfer(\_) 76
- gc\_CRN2LineDev(\_) 79
- GC\_DEST\_OUT\_OF\_ORDER 85
- gc\_Detach(\_) 81
- gc\_DropCall(\_) 84
- gc\_ErrorInfo(\_) 87
- gc\_ErrorValue(\_) 89
  - deprecated function 89
- gc\_Extension(\_) 91
- gc\_GetAlarmConfiguration(\_) 95
- gc\_GetAlarmFlow(\_) 101
- gc\_GetAlarmParm(\_) 104
- gc\_GetAlarmSourceObjectList(\_) 107
- gc\_GetAlarmSourceObjectNetworkID(\_) 110
- gc\_GetANI(\_) 112
  - deprecated function 112
- gc\_GetBilling(\_) 114
- gc\_GetCallInfo(\_) 116
- gc\_GetCallProgressParm(\_) 121
- gc\_GetCallState(\_) 123
- gc\_GetConfigData(\_) 127
- gc\_GetCRN(\_) 134
- gc\_GetCTInfo(\_) 136
- gc\_GetDNIS(\_) 138
  - deprecated function 138
- gc\_GetFrame(\_) 141

gc\_GetInfoElem(\_) 144  
     deprecated function 144  
 gc\_GetLineDev(\_) 147  
     backward compatibility 147  
 gc\_GetLinedevState(\_) 149  
 gc\_GetMetaEvent(\_) 152, 153  
     use with gc\_GetLineDev(\_) 147  
 gc\_GetMetaEventEx(\_) 158  
     use with gc\_GetLineDev(\_) 147  
 gc\_GetNetCRV(\_) 161  
     deprecated function 161  
 gc\_GetNetworkH(\_) 164  
 gc\_GetParm(\_) 166  
 gc\_GetSigInfo(\_) 170  
 gc\_GetUserInfo(\_) 174  
 gc\_GetUsrAttr(\_) 176, 309  
 gc\_GetVer(\_) 178  
 gc\_GetVoiceH(\_) 168, 182  
 gc\_GetXmitSlot(\_) 184  
 gc\_HoldACK(\_) 187  
 gc\_HoldCall(\_) 190  
 gc\_HoldRej(\_) 193  
 GC\_IE\_BLK data structure 387  
 GC\_INFO data structure 388  
 GC\_L2\_BLK data structure 389  
 gc\_LinedevToCCLIBID() 196  
 gc\_Listen(\_) 198  
 gc\_LoadDxParm(\_) 201  
 gc\_MakeCall(\_) 208  
 GC\_MAKECALL\_BLK data structure 390  
 GC\_NETWORK\_CONGESTION 85  
 GC\_NORMAL\_CLEARING 85  
 gc\_Open(\_) 213  
     deprecated function 213  
 gc\_OpenEx(\_) 214  
 GC\_PARM data structure 391  
 GC\_PARM\_BLK data structure 392  
 GC\_PARM\_BLK utility functions, list of 18  
 GC\_PARM\_DATA data structure 393  
 gc\_QueryConfigData(\_) 221  
 GC\_RATE\_U data structure 395  
 gc\_ReleaseCall()  
     deprecated function 226  
 gc\_ReleaseCallEx()  
     use after gc\_DropCall() 85  
 gc\_ReleaseCallEx(\_) 227, 349, 350  
     use with gc\_DropCall(\_) 84  
 GC\_REQ\_CHANNEL\_NOT\_AVAIL 85  
 gc\_ReqANI(\_) 229  
 gc\_ReqMoreInfo(\_) 232  
 gc\_ReqService(\_) 236  
 gc\_ResetLineDev(\_) 239, 350  
 gc\_RespService(\_) 242  
 gc\_ResultInfo(\_) 245  
 gc\_ResultMsg()  
     deprecated function 247  
 gc\_ResultValue()  
     deprecated function 249  
 gc\_RetrieveAck(\_) 251  
 gc\_RetrieveCall(\_) 254  
 gc\_RetrieveRej(\_) 257  
 GC\_RTCM\_EVTDATA data structure 396  
 GC\_SEND\_SIT 85  
 gc\_SendMoreInfo(\_) 260  
 gc\_SetAlarmConfiguration(\_) 263  
 gc\_SetAlarmFlow(\_) 269  
 gc\_SetAlarmNotifyAll(\_) 272  
 gc\_SetAlarmParm(\_) 275  
 gc\_SetBilling(\_) 278  
 gc\_SetCallingNum(\_) 281  
 gc\_SetCallProgressParm(\_) 283  
 gc\_SetChanState(\_) 285  
 gc\_SetConfigData(\_) 288  
 gc\_SetEvtMsk(\_) 294  
 gc\_SetInfoElem(\_) 298  
     deprecated function 298  
 gc\_SetParm(\_) 166, 300  
 gc\_SetUpTransfer(\_) 303  
 gc\_SetUserInfo(\_) 306  
 gc\_SetUsrAttr(\_) 309  
 gc\_SndFrame(\_) 311  
     deprecated function 311  
 gc\_SndMsg()  
     deprecated function 141, 314  
 gc\_Start(\_) 317  
 GC\_START\_STRUCT data structure 397  
 gc\_Stop(\_) 322  
 gc\_StopTrace(\_) 320, 324  
 gc\_StopTransmitAlarms(\_) 326  
 GC\_SUCCESS  
     successful function return value 27  
 gc\_SwapHold(\_) 329  
 gc\_TransmitAlarms(\_) 332  
 GC\_UNASSIGNED\_NUMBER 85  
 gc\_UnListen(\_) 335



GC\_USER\_BUSY 85  
gc\_util\_delete\_parm\_blk(\_) 337  
gc\_util\_find\_parm(\_) 339  
gc\_util\_insert\_parm\_ref(\_) 341  
gc\_util\_insert\_parm\_val(\_) 344  
gc\_util\_next\_parm(\_) 347  
gc\_WaitCall(\_) 240, 349  
GCACT\_ADDMSK 295  
GCACT\_SETMSK 295  
GCACT\_SUBMSK 295  
GCAMS functions, list of 16  
GCEV\_ACCEPT 354  
GCEV\_ACKCALL 354  
GCEV\_ALARM 354  
GCEV\_ALERTING 354  
GCEV\_ALERTING event 295  
GCEV\_ANSWERED 354  
GCEV\_ATTACH 354  
GCEV\_ATTACH\_FAIL 354  
GCEV\_BLINDTRANSFER 354  
GCEV\_BLOCKED 354  
GCEV\_BLOCKED event 295  
GCEV\_CALLINFO 355  
GCEV\_CALLPTOC 355  
GCEV\_CALLSTATUS 355  
GCEV\_COMPLETETTRANSFER 355  
GCEV\_CONGESTION 355  
GCEV\_CONNECTED 355  
GCEV\_D\_CHAN\_STATUS 355  
GCEV\_DETACH 355  
GCEV\_DETACH\_FAIL 355  
GCEV\_DETECTED 356  
GCEV\_DIALING 356  
GCEV\_DIALTONE 356  
GCEV\_DISCONNECTED 85, 356  
GCEV\_DISCONNECTED event  
    gc\_DropCall(\_) 85  
GCEV\_DIVERTED 356  
GCEV\_DROPCALL 356  
GCEV\_ERROR 356  
GCEV\_EXTENSION 357  
GCEV\_EXTENSIONCMPLT 357  
GCEV\_FACILITY 357  
GCEV\_FACILITY\_ACK 357  
GCEV\_FACILITY\_REJ 357  
GCEV\_FATALERROR 357  
GCEV\_GETCONFIGDATA 357  
GCEV\_GETCONFIGDATAFAIL 357  
GCEV\_HOLDACK 358  
GCEV\_HOLDCALL 358  
GCEV\_HOLDCALL event 187  
GCEV\_HOLDREJ 358  
GCEV\_ISDNMSG 358  
GCEV\_L2BFFRFULL 358  
GCEV\_L2FRAME 358  
GCEV\_L2FRAME event 141  
GCEV\_L2NOBRFR 358  
GCEV\_LISTEN 358  
GCEV\_MEDIA\_ACCEPT 359  
GCEV\_MEDIA\_REJECT 359  
GCEV\_MEDIA\_REQ 359  
GCEV\_MEDIADETECTED 359  
GCEV\_MOREDIGITS 359  
GCEV\_MOREINFO 359  
GCEV\_NOFACILITYBUF 359  
GCEV\_NOFACILITYBUF event 118  
GCEV\_NOTIFY 360  
GCEV\_NOUSRINFOBUF 360  
GCEV\_NOUSRINFOBUF event 118  
GCEV\_NSI 360  
GCEV\_OFFERED 56, 350, 360  
GCEV\_OFFERED event 349, 350  
    gc\_CallProgress(\_) 60  
GCEV\_OPENEX 360  
GCEV\_OPENEX\_FAIL 360  
GCEV\_PROCEEDING 360  
GCEV\_PROCESSING 360  
GCEV\_RELEASECALL 360  
GCEV\_RELEASECALL\_FAIL 361  
GCEV\_REQANI 361  
GCEV\_REQMOREINFO 361  
GCEV\_RESETLINEDEV 361  
GCEV\_RESETLINEDEV event 239  
GCEV\_RESTARTFAIL 361  
GCEV\_RETRIEVEACK 361  
GCEV\_RETRIEVECALL 361  
GCEV\_RETRIEVECALL event 251, 257  
GCEV\_RETRIEVEREJ 361  
GCEV\_SENDMOREINFO 362  
GCEV\_SERVICEREQ 362  
GCEV\_SERVICERESP 362  
GCEV\_SERVICERESPCMPLT 362

- GCEV\_SETBILLING 362
- GCEV\_SETCHANSTATE 362
- GCEV\_SETCONFIGDATA 362
- GCEV\_SETCONFIGDATA\_FAIL 363
- GCEV\_SETUP\_ACK 363
- GCEV\_SETUPTRANSFER 363
- GCEV\_STOPMEDIA\_REQ 363
- GCEV\_SWAPHOLD 363
- GCEV\_TASKFAIL 363
- GCEV\_TRANSFERACK 363
- GCEV\_TRANSFERREJ 364
- GCEV\_TRANSIT 364
- GCEV\_UNBLOCKED 364
- GCEV\_UNBLOCKED event 295
- GCEV\_UNLISTEN 364
- GCEV\_USRINFO 364
- GCGLS\_BCHANNEL 149
- GCGLS\_DCHANNEL 149
- GCLIB\_ADDRESS\_BLK data structure 398
- GCLIB\_CALL\_BLK data structure 400
- GCLIB\_CHAN\_BLK data structure 401
- GCLIB\_MAKECALL\_BLK data structure 402
- GCME\_GC\_EVENT bit 152
- GCPR\_CALLINGPARTY 301
- GCSR functions, list of 18
- Global Call Alarm Management System functions, list of 16
- Global Call error codes
  - retrieval 89

## I

- IE (Information Element) 118
- in-band tones 60
- Information Element (IE) 118
- information elements
  - setting 298
- ISDN
  - setting information elements 298
- ISDN\_BN 229
- ISDN\_BN\_PREF 229
- ISDN\_CA\_TSC 229
- ISDN\_CPN 229
- ISDN\_CPN\_PREF 229
- ISDN-Specific functions, list of 15

## L

- LAPD protocol 314

- Layer 2 access
  - enabling 141
- library identification code 65
- Library Information functions, list of 12

## M

- media resources
  - detaching 81
- METAEVENT data structure 154, 403
  - use in gc\_GetCRN(\_) 134

## N

- Network Specific Facility IE 118
- notification event
  - definition 353
- nr\_scroute(\_) 164
- Null state 350

## O

- Optional Call Handling functions, list of 12

## P

- Parameter ID definitions, info\_id 116

## R

- resource
  - attaching 49
- resourceh device handle 49
- result values
  - definition 407
- retrieving event information 152
- RTCM functions, list of 18
- Run Time Configuration Management functions, list of 18

## S

- SC\_TSINFO data structure 405
- Service Request functions, list of 18
- service state of line 285
- setup ACK message 363
- SIT (Special Information Tone) 60
- Special Information Tone 85
- Special Information Tone (SIT) 60
- SRL device handle 164
- SRL device handles 81



- SRL events 152
- supervised transfer 329
- syntax convention for functions 27
- System Controls and Tools functions, list of 13

## T

- termination event
  - definition 353
- timeout 350

## U

- U\_IES 118
- unsolicited event
  - definition 353
- user-to-user information 298
- User-to-User Information (UII) 118
- usrattr 309
- usrattr parameter 214
- UII (User-to-User Information) 118

## V

- Vari-A-Bill service 114, 278
- Voice and Media functions, list of 15

