intel®

Global Call API for Host Media Processing

Library Reference

August 2005

05-2362-003

Publication Date: August 2005

Document Number: 05-2362-003

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 Telecom and Compute Products website at:
http://www.intel.com/design/network/products/telecom

For **Sales Offices** and other contact information, visit the Buy Telecom Products page at:
http://www.intel.com/buy/networking/telecom.htm

Contents

Figures

Tables

Revision History

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

Document No.	Publication Date	Description of Revisions
05-2362-003	August 2005	Function Summary by Category chapter: Added new section for Third-Party Call Control Functions. Function Information chapter: Added function reference page for gc_SipAck( ). gc_CCLibNameToID( ) function reference: Added “GC_H3R_LIB” (IP call control library) to the list of call control library names. gc_Extension( ) function reference: Added caution about how to identify the extension function across technologies. gc_Start( ) function reference: Added caution that the network adapter must be enabled before calling the function when using Global Call over IP, and added information about how to start with the network adapter disabled. CCLIB_START_STRUCT and GC_CCLIB_STATE data structure reference: Added “GC_H3R_LIB” (IP call control library) to the list of call control library names. GCLIB_MAKECALL_BLK data structure reference: Corrected typo in the Description section; changed GC**MAKE**CALLBLK_DEFAULT to GC**MK**CALLBLK_DEFAULT.
05-2362-002	April 2005	Function Summary by Category chapter: Added gc_SetAuthenticationInfo( ) to Feature Transparency and Extension (FTE) Functions. Added gc_util_copy_parm_blk( ), gc_util_find_parm_ex( ), gc_util_insert_parm_ref_ex( ), and gc_util_next_parm_ex( ) to GC_PARM_BLK Utility Functions. Function Information chapter: Added the following function reference pages: gc_SetAuthenticationInfo( ) gc_util_copy_parm_blk( ) gc_util_find_parm_ex( ) gc_util_insert_parm_ref_ex( ) gc_util_next_parm_ex( ) Data Structures chapter: Added GC_PARM_DATA_EXT data structure reference page. Error Codes chapter: Made minor change to description of GCRV_CCLIBSPECIFIC.
05-2362-001	September 2004	Initial version of document. Much of the information contained in this document was previously published in the *Global Call API for Linux and Windows Operating Systems Library Reference*, document number 05-1816-005.

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 information about the Global Call API that is supplied with the Intel NetStructure® Host Media Processing (HMP) product, including function descriptions, events, data structures, and error codes. Supplemental information about using the Global Call API with HMP is provided in the *Global Call IP for Host Media Processing Technology Guide*.

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

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

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

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 the various categories of functions in the Global Call API library and provides a brief description of each function.
- Chapter 2, "Function Information" provides an alphabetical reference to the Global Call API functions that are used with HMP.

  Note: The purpose of the code examples in this chapter is to illustrate how Global Call functions are used. The examples are not necessarily for HMP-specific applications.
- Chapter 3, "Events" provides an alphabetical reference to the events used by the Global Call software with HMP.
- Chapter 4, "Data Structures" provides an alphabetical reference to the data structures used by the functions in the Global Call API library with HMP.
- 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:

- For guidelines on building applications using the Global Call API, see the *Global Call API Programming Guide*.
- For Global Call IP-specific information, see the *Global Call IP for Host Media Processing Technology Guide*.
- For details on the Standard Runtime Library (SRL), supported programming models, and programming guidelines for building all applications, see the *Standard Runtime Library API Programming Guide*. The SRL is 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.
- For details on all functions and data structures in the SRL, see the *Standard Runtime Library API Library Reference*.
- For information about the software release, system requirements, features, and documentation, see the Release Guide for the software release you are using.
- For details on known problems and late-breaking updates or corrections to the release documentation, see the Release Update.

  Be sure to check the Release Update for the software release you are using for any updates or corrections to this publication. Release Updates are available on the Telecom Support Resources website at *http://www.intel.com/design/network/products/telecom/software/index.htm*
- For technical support, see *http://developer.intel.com/design/telecom/support/*. This website contains developer support information, downloads, release documentation, technical notes, application notes, a user discussion forum, and more.

Function Summary by Category 1

This chapter describes the categories into which the Global Call API library functions can be logically grouped. The topics in this chapter are:

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_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 that are not provided by the basic Global Call functions. These functions are used to interface with all signaling systems.

gc_AcceptCall( )
indicates to the originator that the call will be answered

gc_CallAck( )
indicates call reception and optionally takes action

gc_GetCallInfo( )
retrieves information associated with a call

gc_GetVer( )
retrieves the version number of a specified software component

1.4 Supplementary Service Functions

The following functions are used to transfer calls for specific technologies.

gc_AcceptInitXfer( )
accept request to initiate a supervised transfer

gc_AcceptXfer( )
accept call transfer request

gc_InitXfer( )
initiate a supervised transfer

gc_InvokeXfer( )
request a blind or supervised call transfer

gc_RejectInitXfer( )
reject request to initiate a supervised transfer

gc_RejectXfer( )
reject call transfer request

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_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_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_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_SetConfigData( )
updates the configuration data

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_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

1.7 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

1.8 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_SetAuthenticationInfo( )
sets IP authentication information

gc_SetUserInfo( )
permits the specification of technology-specific user information

1.9 Real Time Configuration Management (RTCM) Functions

The following function is used to update configuration parameter data dynamically.

gc_SetConfigData( )
updates the configuration data

1.10 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

1.11 Third-Party Call Control Functions

The following function is used only with IP (SIP), technology third party call control (3PCC) mode.

gc_SipAck( )
acknowledge a SIP 200OK message in 3PCC mode

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_copy_parm_blk( )
copies the specified GC_PARM_BLK

gc_util_delete_parm_blk( )
deletes the specified GC_PARM_BLK

gc_util_find_parm( ), **gc_util_find_parm_ex( )**
finds a parameter in a GC_PARM_BLK. Use **gc_util_find_parm_ex( )** if the parameter data can potentially exceed 255 bytes (SIP message headers, for example).

gc_util_insert_parm_ref( ), **gc_util_insert_parm_ref_ex( )**
adds a parameter by reference to a GC_PARM_BLK. Use **gc_util_insert_parm_ref_ex( )** if the parameter data can potentially exceed 255 bytes (SIP message headers, for example).

gc_util_insert_parm_val( )
adds a parameter by value to a GC_PARM_BLK

gc_util_next_parm( ), **gc_util_next_parm_ex( )**
finds the next parameter in a GC_PARM_BLK. Use **gc_util_next_parm_ex( )** if the parameter data can potentially exceed 255 bytes (SIP message headers, for example).

Function Information 2

This chapter provides an alphabetical reference to the functions in the Global Call API library that are used with Host Media Processing (HMP). A general description of the function syntax convention is provided before the detailed function information. For some functions, you are directed to see the *Global Call IP for Host Media Processing Technology Guide* for additional, more detailed technology-specific information.

Note: The purpose of the code examples in this chapter is to illustrate how Global Call functions are used. The examples are not necessarily for HMP-specific applications.

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

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: optional call handling

Mode: asynchronous

■ 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.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
crn	call reference number
rings	specifies how long (the number of rings) the protocol handler will wait before notifying the calling entity. Valid values are: • 0 to 14 – the call is accepted after the specified number of rings • 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 **gc_AnswerCall( )**. For protocols not using rings, the **rings** parameter is ignored.
mode	set to EV_ASYNC for asynchronous 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>
#include <gcip.h>
#include <gcip_defs.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_AcceptInitXfer( )

Name: int gc_AcceptInitXfer(crn, reroutinginfop, mode)

Inputs:

CRN crn	• call reference number for the call between remote transferring party A and transferred-to party C
GC_REROUTING_INFO * reroutinginfop	• pointer to the rerouting information associated with this acceptance
unsigned long mode	• async

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: supplementary service - call transfer

Mode: asynchronous

■ Description

The **gc_AcceptInitXfer( )** function is called by the transferred-to party (party C) to accept the initiate transfer request from the remote transferring party (party A), indicating that party C is willing to participate in the call transfer. Prior to calling this function, party C had to be notified of the initiate transfer request via an unsolicited event GCEV_REQ_INIT_XFER.

The remote transferring party A is notified of the acceptance via a GCEV_INIT_XFER event.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
crn	call reference number for the call between the remote transferring party A and the transferred-to party C that received the initiate transfer request
parmblkp	points to the GC_REROUTING_INFO structure. The structure lists the data needed for acceptance. The user-defined rerouting number and extra data go here. This parameter is optional and should be set to 0 if not used. In this function, party C can provide its rerouting address through the **GC_REROUTING_INFO** parameter or can use the CCLib default rerouting address.
mode	set to EV_ASYNC for asynchronous execution

Regardless of whether the **gc_AcceptInitXfer( )** function succeeds or fails, the call state of party C returns to its original call state (GCST_CONNECTED).

Preceding Events

GCEV_REQ_INIT_XFER
: unsolicited event, notifies application at party C of the initiate transfer request from remote party A

Termination Events

GCEV_ACCEPT_INIT_XFER
: indicates that the **gc_AcceptInitXfer( )** function was successful, that is, party C successfully accepted the initiate transfer request from remote party A

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

Cautions

The **gc_AcceptInitXfer( )** function can be called only when the call is in the GCST_REQ_INIT_XFER call state.

Errors

If this function returns <0 to indicate failure, use the **gc_ErrorInfo( )** function for error information. If the GCEV_ACCEPT_INIT_XFER_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>
#include <gcip.h>
#include <gcip_defs.h>

#define MAX_CHAN 30               /* maximum number of channels in system */
/*
* Data structure which stores all information for each line
*/
static struct linebag {
    LINEDEV    ldev;               /* Global Call API line device handle */
    CRN        crn;                /* Global Call API call handle */
    CRN        consultation_crn;   /* Global Call API call handle */
    int        blocked;            /* channel blocked/unblocked */
} port[MAX_CHAN+1];
struct linebag *pline;             /* pointer to access line device */
```

```
/*
 * Assume the following have been done:
 *    1. Opened line devices for each time slot on the network interface board.
 *    2. Each line device is stored in linebag structure "port".
 *    3. The original (primary) call has been established between Transferring party A and
 *       Transferred party B and the call is in connected or on hold state.
 *    4. The party C received the GCEV_REQ_INIT_XFER.
 */

int accept_transferinitiate(int port_num)
{
    GC_INFO        gc_error_info;    /* Global Call error information data */

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

    /* Accept the call transfer request */
    if (gc_AcceptInitXfer(pline->crn, NULL, EV_ASYNC) == -1)
    {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_AcceptInitXfer() 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_AcceptXfer( )**
- **gc_InitXfer( )**
- **gc_InvokeXfer( )**
- **gc_RejectInitXfer( )**
- **gc_RejectXfer( )**

gc_AcceptXfer( )

Name: int gc_AcceptXfer(crn, parmblkp, mode)

Inputs:

CRN crn	• call reference number for the call between remote transferring party A and the local party receiving the call transfer request
GC_PARM_BLK *parmblkp	• pointer to the parameter block associated with the acceptance
unsigned long mode	• async

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: supplementary service - call transfer

Mode: asynchronous

Description

The **gc_AcceptXfer( )** function is called by the local party (transferred party B or transferred-to party C) to accept the call transfer request from the remote transferring party A. Prior to calling this function, the local party had to be notified of the call transfer request via an unsolicited event GCEV_REQ_XFER, which provided the rerouting information.

The remote transferring party A may or may not be notified of the acceptance, depending on the technology and protocol. For applications using H.323/H.450.2, the remote transferring party A will not receive notification of the acceptance. For applications using SIP, the remote transferring party A can optionally receive notification of the acceptance via a GCEV_INVOKE_XFER_ACCEPTED event.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
crn	call reference number for the call between the remote transferring party A and the local party that received the call transfer request
parmblkp	ignored for IP; set to NULL
mode	set to EV_ASYNC for asynchronous execution

After successfully accepting the request, the local party will make an outbound rerouting call for a transfer. The application calls the **gc_MakeCall( )** function to make the call between party B and C for this transfer using the calling address provided in GCEV_REQ_XFER.

When receiving the GCEV_REQ_XFER event, the application can obtain the information for the rerouting number or address (a GC_REROUTING_INFO data structure) from extevtdatap in the METAEVENT data structure. Because the extevtdtatap is managed by Global Call (that is, Global Call owns this data memory), the user application does not need to allocate or deallocate memory for extevtdatap. If the data needs to be reserved for future use (for example, making a rerouting call), the user should save the data, including all fields, to its own memory space.

When calling **gc_MakeCall( )** to make a rerouting outbound call, the user application must provide the CRN for the primary call additional information in the GC_MAKECALL_BLK, e.g., the CRN for the primary call in blind transfer, and the CRN for the primary and secondary call in supervised transfer. This is passed via the GC_PARM_BLK within the GCLIB_MAKECALL_BLK using a set ID of GCSET_SUPP_XFER and a parameter ID of GCPARM_PRIMARYCALL_CRN.

For the inbound-side party receiving the rerouting call, the GCRV_XFERCALL result value will be associated with the GCEV_DETECTED/GCEV_OFFERED event to indicate the transfer call. Extra data may also be associated with these two events through extevtdatap of METAEVENT.

■ Preceding Events

GCEV_REQ_XFER
: unsolicited event, notifies application at the local party receiving the call transfer request from remote party A

■ Termination Events

GCEV_ACCEPT_XFER
: indicates that the **gc_AcceptXfer( )** function was successful, that is, the local party successfully accepted the call transfer request from remote party A

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

■ Other Related Events

GCEV_XFER_CMPLT
: unsolicited event, notifies application at the local party accepting the call transfer request that the call transfer was completed (that is, the call between party B and C was connected)

GCEV_XFER_FAIL
: unsolicited event, notifies application at the local party accepting the call transfer request that the call transfer was not completed because of a failure (time-out, busy, no answer, etc.)

■ Cautions

The **gc_AcceptXfer( )** function can be called only when the call to be transferred is in the GCST_REQ_XFER call state (that is, after receiving GCEV_REQ_XFER).

■ Errors

If this function returns <0 to indicate failure, use the **gc_ErrorInfo( )** function for error information. If the GCEV_XFER_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, namely *gcip_defs.h* or *ccerr.h* for the IP call control library.

Additional result values for GCEV_XFER_FAIL include:

GCRV_LOCALPARTY_PROT_TIMEOUT
: local party protocol time-out

GCRV_REMOTEPARTY_PROT_TIMEOUT
: remote party A protocol time-out

Example

```
#include <stdio.h>
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcip.h>
#include <gcip_defs.h>
#include <gccfgparm.h>

#define MAX_CHAN 30               /* maximum number of channels in system */
/*
* Data structure which stores all information for each line
*/
static struct linebag {
    LINEDEV    ldev;               /* Global Call API line device handle */
    CRN        crn;                /* Global Call API call handle */
    CRN        consultation_crn;   /* Global Call API call handle */
    int        blocked;            /* channel blocked/unblocked */
} port[MAX_CHAN+1];
struct linebag *pline;             /* pointer to access line device */
/*
 * Assume the following have been done:
 *    1. Opened line devices for each time slot on the network interface board.
 *    2. Each line device is stored in linebag structure "port".
 *    3. The original (primary) call has been established between Transferring party A and
 *       Transferred party B and the call is in connected or answered state.
 *    4. The party B received the GCEV_REQ_XFER.
 */
int accept_calltransfer(int port_num)
{
    GC_INFO       gc_error_info;    /* Global Call error information data */

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

    /* Accept the call transfer request */
    if (gc_AcceptXfer(pline->crn, NULL, EV_ASYNC) == -1)
    {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_AcceptXfer() 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_AcceptXfer( )**
- **gc_InitXfer( )**
- **gc_InvokeXfer( )**
- **gc_RejectInitXfer( )**
- **gc_RejectXfer( )**

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

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.

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>
#include <gcip.h>
#include <gcip_defs.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

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
metaeventp	points to the metaevent
alarm_number	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>
#include <gcip.h>
#include <gcip_defs.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 nunber is %d\n", alarm_number);
    }
```

See Also

- **gc_AlarmName( )**
- **gc_AlarmNumberToName( )**

gc_AlarmNumberToName( )

Name: int gc_AlarmNumberToName(aso_id, alarm_number, name)

Inputs:

unsigned long aso_id	• alarm source object ID
long alarm_number	• alarm number
char **name	• 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

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
aso_id	alarm source object (ASO) ID
alarm_number	alarm number to be converted
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 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 */
#include <gcip.h>
#include <gcip_defs.h>

    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

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
metaeventp	points to metaevent
aso_id	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>
#include <gcip.h>
#include <gcip_defs.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

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
aso_id	alarm source object ID
aso_name	points to the location where the pointer to the alarm source object name is to be stored. The **aso_name** 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>
#include <gcip.h>
#include <gcip_defs.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

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
metaeventp	points to metaevent
aso_name	points to the location where the pointer to the alarm source object name is to be stored. The **aso_name** 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>
#include <gcip.h>
#include <gcip_defs.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

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
aso_name	points to the location of ASO name. The **aso_name** cannot be NULL.
aso_id	points to the destination where the alarm source object ID is returned. The **aso_id** 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>
#include <gcip.h>
#include <gcip_defs.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:

CRN crn	• call reference number
int rings	• number of rings before return
unsigned long mode	• async

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: basic

Mode: asynchronous

■ 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.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
crn	call reference number
rings	specifies the number of rings the protocol handler waits before notifying the calling entity. The valid values are as follows: • 0 to 14 – the call is answered after the specified number of rings • 15 or greater – the call will ring forever; this allows the application to accept calls without answering them For protocols not using rings, the **rings** parameter is ignored.
mode	set to EV_ASYNC for asynchronous 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>
#include <gcip.h>
#include <gcip_defs.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_AttachResource( )

Name: int gc_AttachResource (linedev, resourceh, resourceattrp, retblkpp, resourcetype, mode)

Inputs:

LINEDEV linedev	• Global Call line device handle
int resourceh	• voice or media device handle
GC_PARM_BLKP resourceattrp	• pointer to voice or media attribute information
GC_PARM_BLKP* retblkpp	• pointer to address of GC_PARM_BLK structure where voice or media attribute information is to be returned
int resourcetype	• type of resource
unsigned long mode	• async

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: voice and media

Mode: asynchronous

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, **resourceh**, will be used to handle related Global Call functions requiring a voice or media resource for that line device.

Parameter	Description
linedev	Global Call line device handle
resourceh	SRL device handle for the voice or media resource that is to be attached to the line device
resourceattrp	points to the GC_PARM_BLK structure containing attributes of the voice or media resource being specified by **resourceh**.
retblkpp	points to the address of GC_PARM_BLK structure where voice or media attribute information is to be returned
resourcetype	type of resource. Possible values are: • GC_VOICEDEVICE • GC_MEDIADEVICE
mode	set to EV_ASYNC for asynchronous execution

Termination Events

GCEV_ATTACH
: indicates successful completion of the **gc_AttachResource( )** function.

GCEV_ATTACH_FAIL
: indicates that the **gc_AttachResource( )** function failed.

Cautions

- 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.

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 <dxxxlib.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcip.h>
#include <gcip_defs.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("dxxxB1C1", NULL);
        if (voiceh != -1) {
            if (gc_AttachResource(ldev, voiceh, NULL, NULL, GC_VOICEDEVICE,
                EV_ASYNC) == 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_GetResourceH( )**
- **gc_OpenEx( )**

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

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: optional call handling

Mode: asynchronous

■ 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.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
crn	call reference number
callack_blkp	points to the GC_CALLACK_BLK structure where the type field specifies the type of service requested by the **gc_CallAck( )** function. See GC_CALLACK_BLK, on page 252 for further information.
mode	set to EV_ASYNC for asynchronous execution

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 1 shows the values that are supported for the type field.

Table 1. 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. **Note:** If this value is used for type, then the 'info' structure must be used in the service union. See Table 2 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.
GCACK_SERVICE_DNIS	Request retrieval of additional DNIS digits. **Note:** This type is not supported for all call libraries. It is recommended that GCACK_SERVICE_PROC be used instead.

Table 2 describes the fields in the 'info' structure used when GCACK_SERVICE_INFO is specified.

Table 2. Fields in the 'info' Structure for GCACK_SERVICE_INFO

Field	Description
info_type	Type of information requested. The possible values are: • DESTINATION_ADDRESS – request more DNIS • ORIGINATION_ADDRESS – request more ANI
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.

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 <memory.h>
#include <gclib.h>
#include <gcerr.h>
#include <gcisdn.h>
#include <gcip.h>
#include <gcip_defs.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_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

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
cclibid	identification number of call control library. If a library name is not associated with this parameter, then the library name will be NULL.
lib_name	points to the location where the name of the call control library associated with the **cclibid** parameter is stored. See **gc_CCLibNameToID( )** 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>
#include <gcip.h>
#include <gcip_defs.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

Description

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

Parameter	Description
lib_name	name of the call control library whose library ID is to be retrieved. Valid call control library names are: • "GC_ALL_LIB" – All call control libraries • "GC_CUSTOM1_LIB" – Custom call control library 1 • "GC_CUSTOM2_LIB" – Custom call control library 2 • "GC_DM3CC_LIB – DM3CC call control library • "GC_H3R_LIB" – IP 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
cclibidp	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>
#include <gcip.h>
#include <gcip_defs.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%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %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_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: gclib.h
gcerr.h

Category: library information

Mode: synchronous

■ 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
cclib_name	name of the call control library. See **gc_CCLibNameToID( )** for valid call control library names. The string must be set to one of these names and terminated by a NULL.
cclib_infop	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: • GC_CCLIB_AVAILABLE – available library (started successfully) • GC_CCLIB_CONFIGURED – configured library • GC_CCLIB_FAILED – library failed to start

■ 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>
#include <gcip.h>
#include <gcip_defs.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%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %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: gclib.h
gcerr.h

Category: library information

Mode: synchronous

■ 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
cclib_status	points to the GC_CCLIB_STATUS structure, which contains the library status information. See GC_CCLIB_STATUS, on page 255 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>
#include <gcip.h>
#include <gcip_defs.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%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);

    }
    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: gclib.h
gcerr.h

Category: library information

Mode: synchronous

■ 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
cclib_name	call control library name. See **gc_CCLibNameToID( )** for valid call control library names. The string must be set to one of these names in uppercase and terminated by a NULL.
cclib_infop	points to an integer location for status information. The possible states are: • GC_CCLIB_AVAILABLE – available library (started successfully) • GC_CCLIB_CONFIGURED – configured library • GC_CCLIB_FAILED – library failed to start If the status of all the libraries is to be retrieved, that is, if GC_ALL_LIB is specified by the **cclib_name** parameter, the application must pass the pointer to a GC_CCLIB_STATUSALL structure. The status of all the libraries will be returned in this structure.

■ 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_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>
#include <gcip.h>
#include <gcip_defs.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 - %s, CC ErrorValue: 0x%lx - %s\n",
               "GC_ALL_LIB", 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);

    }

    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

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( )**.

Parameter	Description
linedev	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>
#include <gcip.h>
#include <gcip_defs.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_Detach( )**
- **gc_OpenEx( )**

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

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
crn	call reference number
linedevp	points to the location where the output LINEDEV identification code will be stored. The line device is created when the function **gc_OpenEx( )** 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>
#include <gcip.h>
#include <gcip_defs.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%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);
    }
    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

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: voice and media

Mode: asynchronous

■ 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.

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

■ Termination Events

GCEV_DETACH
indicates successful completion of the **gc_Detach( )** function.

GCEV_DETACH_FAIL
indicates that the **gc_Detach( )** function failed.

■ 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>
#include <gcip.h>
#include <gcip_defs.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_iptB1T1:P_IP:M_ipmB1C1:V_dxxxB1C1"
 *    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_GetResourceH(ldev, GC_VOICEDEVICE, &voiceh) == GC_SUCCESS) {
        if (gc_Detach(ldev, voiceh, EV_ASYNC) != 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_GetResourceH() error */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_GetResourceH() 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

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: basic

Mode: asynchronous

■ 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.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
crn	call reference number
cause	indicates reason for disconnecting or rejecting a call. See Table 3 for a list of possible causes, and see the *Global Call IP for Host Media Processing Technology Guide* for valid and/or additional causes for your specific technology.
mode	set to EV_ASYNC for asynchronous execution

Table 3. 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_USER_BUSY	End user is busy
† See the *Global Call IP for Host Media Processing Technology Guide* 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( )**.
- Different technologies and protocols support some or all of the cause values defined above; see the *Global Call IP for Host Media Processing Technology 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>
#include <gcip.h>
#include <gcip_defs.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_DROPCALL 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

Description

The **gc_ErrorInfo( )** function provides error information about a failed function. 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. 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
a_Infop	points to the GC_INFO 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_Info);

void PrintErrorInfo(void)
{
    int                retCode;
    GC_INFO            t_Info;
    retCode = gc_ErrorInfo(&t_Info);
    if (retCode == GC_SUCCESS) {
        printf("gc_ErrorInfo() successfully called\n");
        PrintGC_INFO(&t_Info);
    } 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_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_ErrorValue( )** (deprecated)
- **gc_ResultInfo( )**
- **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: gclib.h
gcerr.h

Category: system controls and tools

Mode: synchronous

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. Each of these call control library time-out errors are mapped to EGC_TIMEOUT. However, the specific time-out error detected by the call control library will be available through **cclib_errorp**.

Parameter	Description
gc_errorp	points to the location where the Global Call error code will be stored
cclibidp	points to the location to store the identification number of the call control library where the error occurred
cclib_errorp	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 <gclib.h>
#include <gcerr.h>
#include <gcip.h>
#include <gcip_defs.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) {
        printf("Error code=%d\n",cc_error);
    } else {
        printf("Could not get error value\n");
    }
}
```

See Also

- **gc_ErrorInfo( )**
- **gc_ResultInfo( )**
- **gc_CCLibIDToName( )**

gc_Extension( )

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

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
unsigned char ext_id	• extension function identifier as specified by the technology
GC_PARM_BLK* parmblkp	• pointer to GC_PARM_BLK as specified by the technology
GC_PARM_BLKP* retblkp	• pointer to address of GC_PARM_BLK union where parameter values are to be returned. Optional depending on technology.
unsigned long mode	• async

Returns: 0 if successful
<0 if failure

Includes: gclib.h

Category: FTE

Mode: asynchronous

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 287 for more information.

The *Global Call IP for Host Media Processing Technology Guide* provides 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.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
target_type	target object type. Valid types are: • GCTGT_GCLIB_CHAN • GCTGT_GCLIB_CRN See Table 14, "Global Call Parameter Entry List Maintained in GCLIB", on page 291 for details about these two types.
target_id	target object identifier. This identifier, along with **target_type**, uniquely specifies the target object. Valid identifiers are: • line device handle • call reference number
ext_id	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 *Global Call IP for Host Media Processing Technology Guide* for actual definitions.
parmblkp	points to GC_PARM_BLK as defined by the technology. This parameter is not used in cases where extension functions do not have any parameters.
retblkp	points to the GC_PARM_BLK 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. 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.
mode	set to EV_ASYNC for asynchronous 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( ).**
- Upon receiving a GCEV_EXTENSION event, the application must use a combination of technology (based on **target_id**) and **ext_id** to uniquely identify the extension function across technologies.

- The GC_PARM_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

The *Global Call IP for Host Media Processing Technology Guide* contains technology-specific examples of using the **gc_Extension( )** function.

See Also

None

gc_GetAlarmConfiguration( )

Name: int gc_GetAlarmConfiguration(linedev, aso_id, alarm_list, alarm_config_type)

Inputs:

LINEDEV linedev	• Global Call line device handle
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

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
linedev	Global Call line device handle. 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 **gc_SetAlarmConfiguration( )** function.
aso_id	ASO ID. Use the **gc_AlarmSourceObjectNameToID( )** 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 **aso_id** not associated with a line device, NULL can be used.

Parameter	Description
alarm_list	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 ALARM_LIST structure provide the following information: • n_alarms – contains the number of alarms in the list • alarm_fields – an array of data structures that contain information about the list of alarms The following information is contained in the ALARM_FIELD data structures for each alarm in the list: • alarm_number – contains the alarm number • alarm_data – contains the value of the information that was requested, for example, ALARM_NOTIFY or ALARM_NONOTIFY if **alarm_config_type** = ALARM_CONFIG_NOTIFY. The data type of alarm_data is dependent upon the **alarm_config_type** parameter. See Table 4 for data types. See ALARM_LIST, on page 244 and ALARM_FIELD, on page 243 for more information about those data structures.
alarm_config_type	alarm configuration type. See Table 4 for the possible values. The information is returned in the data field in **alarm_list** for all alarms in the specified alarm source object.

Table 4. 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 4. 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 **alarm_list**. (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>
#include <gcip.h>
#include <gcip_defs.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_GetAlarmConfiguation() 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_GetAlarmConfiguation() 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

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
aso_id	alarm source object (ASO) ID. Use the **gc_AlarmSourceObjectNameToID( )** 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 **aso_id** not associated with a line device, NULL can be used.
flow	points to the destination for the current flow control value. Possible values for flow control are: • ALARM_FLOW_ALWAYS – all alarms with notification on are sent to the application (default) • ALARM_FLOW_ALWAYS_BLOCKING – only blocking alarms with notification on are sent to the application. Non-blocking alarms are not sent. • 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. • 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)

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>
#include <gcip.h>
#include <gcip_defs.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	• Global Call line device handle
	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	

■ 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).

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
linedev	Global Call line device handle
aso_id	alarm source object (ASO) ID. Use the **gc_AlarmSourceObjectNameToID( )** 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.
ParmSetID	ID of the parameter set as defined by the ASO. See the *Global Call API Programming Guide* for a list of the Global Call-aware ASO IDs and the *Global Call IP for Host Media Processing Technology Guide* for technology-specific ASO IDs.
alarm_parm_list	points to the alarm parameter list. See ALARM_PARM_LIST, on page 246 for more information.
mode	set to EV_SYNC 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 *Global Call IP for Host Media Processing Technology 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 <gclib.h>
#include <gcerr.h>
#include <dtilib.h>                   /* for ASO symbols */
#include <gcip.h>
#include <gcip_defs.h>

    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:	LINEDEV linedev	• Global Call line device handle
	int ByNameOrById	• specifies whether list is a list of ASO names or a list of ASO IDs
	ALARM_SOURCE_OBJECT_LIST *aso_list	• pointer to alarm source object list
Returns:	0 if successful <0 if failure	
Includes:	gclib.h gcerr.h	
Category:	GCAMS	
Mode:	synchronous	

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
linedev	Global Call line device handle
ByNameOrById	specifies whether the function is to return a list of ASO names or a list of ASO IDs. Valid values are: • ASO_LIST_BY_NAME • ASO_LIST_BY_ID
aso_list	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 aso_data field of ALARM_SOURCE_OBJECT_FIELD is of type char * if the request is "by name". If the request is by ID, the field is typed an unsigned long. See ALARM_SOURCE_OBJECT_LIST, on page 248 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>
#include <gcip.h>
#include <gcip_defs.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%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
    {
        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%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
        {
        /*
         -- 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	• Global Call line device handle
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

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
linedev	Global Call line device handle
aso_networkID	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>
#include <gcip.h>
#include <gcip_defs.h>

void gc_demo_open_E1orT1_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%hx - %s,
                CCLibID: %i - %s, CC ErrorValue: 0x%lx - %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%lx,
                GC ErrorValue: 0x%hx - %s, CCLibID: %i - %s, CC ErrorValue: 0x%lx - %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_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 gcerr.h	
Category:	optional call handling	
Mode:	synchronous	

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.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
crn	call reference number
info_id	ID of the information to be collected. See the *Global Call IP for Host Media Processing Technology Guide*.
valuep	buffer address where the requested information is stored

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. 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_iptB1T1:P_IP:M_ipmB1C1:V_dxxxB1C1, 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>
#include <gcip.h>
#include <gcip_defs.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_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

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
crn	call reference number
state_ptr	points to the location where the call state value will be returned. Possible state values are: • GCST_ACCEPTED – An inbound call was accepted; the call is in the Accepted state. • GCST_ALERTING – The call is waiting for the destination party to answer; the call is in the Alerting state (alerted sent or received). • GCST_CALLROUTING – Exists for an incoming call when the user has sent an acknowledgment that all call information necessary to effect call establishment has been received. • GCST_CONNECTED – An inbound or outbound call was connected; the call is in the Connected state. • GCST_DETECTED – An incoming call has been received but has not yet been offered to the application; the call is in the Detected state. • GCST_DIALING – An outbound call request was received; the call is in the Dialing state.

Parameter	Description
	• GCST_DISCONNECTED – The call was disconnected from the network, the call is in the Disconnected state. • GCST_IDLE – The call is not active; the call is in the Idle state. • GCST_NULL – The call was released; the call is in the Null state. • GCST_OFFERED – An inbound call was received, the call is in the Offered state. • 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.

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.

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 <gcip.h>
#include <gcip_defs.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 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);
       }
   }

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

■ See Also

None

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

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
crn_ptr	points to the memory address where the call reference number is stored
metaeventp	points to the METAEVENT structure filled in by the **gc_GetMetaEvent( )** or **gc_GetMetaEventEx( )** function. See METAEVENT, on page 273 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>
#include <gcip.h>
#include <gcip_defs.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_devinfop)

Inputs: LINEDEV linedev • Global Call line device

CT_DEVINFO *ct_devinfop • pointer to device information structure

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: system controls and tools

Mode: synchronous

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 *ctinfo.h*, which is referenced by *gclib.h*.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
linedev	Global Call device handle
ct_devinfop	points to the CT_DEVINFO device information structure. See CT_DEVINFO, on page 250 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>
#include <gcip.h>
#include <gcip_defs.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_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

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
linedevp	points to the location where the output LINEDEV is stored
metaeventp	points to the METAEVENT structure filled in by the **gc_GetMetaEvent( )** or the **gc_GetMetaEventEx( )** 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>
#include <gcip.h>
#include <gcip_defs.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_GetLineDev() 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);
    }

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

See Also

- **gc_GetCRN( )**
- **gc_GetMetaEvent( )**
- **gc_GetMetaEventEx( )** (Windows extended asynchronous model only)

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

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 273 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 METAEVENT 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_GetMetaEvent( )** 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>
#include <gcip.h>
#include <gcip_defs.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>
#include <gcip.h>
#include <gcip_defs.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           evttype;          /* type of event */

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

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

        /* Get the next event */
        if (gc_GetMetaEvent(&metaevent) != GC_SUCCESS)
        {
            /* process error return */
            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 NULL_STATE:
                    switch (evttype) {
                        case GCEV_OFFERED:
                            pline -> state = OFFERED_STATE;
                            accept_call();
                            break;
                       default:
                            /* handle other events here */
                       break;
                    }
                break;

                case OFFERED_STATE:
                    switch (evttype) {
                        case GCEV_ACCEPT:
                            pline -> state = ACCEPTED_STATE;
                            answer_call();
                            break;
                        default:
                            /* handle other events here */
                            break;
                    }
                break;

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

■ See Also

- **gc_GetLineDev( )**
- **gc_GetMetaEventEx( )** (Windows extended asynchronous model only)
- **gc_ResultInfo( )**

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: gclib.h
gcerr.h

Category: system controls and tools

Mode: synchronous

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
metaeventp	points to the METAEVENT structure to be filled in by this function. See METAEVENT, on page 273 for more information.
evt_handle	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;
hdls[ hdlcnt++ ] = GetGlobalCallHandle();
hdls[ hdlcnt++ ] = GetVoiceHandle();

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

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

    if (rc != SR_TMOUT)
    {
        /*
         *  Update
         */
        rc = gc_GetMetaEventEx(&g_Metaevent, evtHdl);
        if (rc != GC_SUCCESS)
        {
            /* 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_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

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.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
linedev	Global Call line device handle. This can be one of the following types: • A non-IP media Global Call line device handle • An IP media Global Call line device handle
resourcehp	address at which the associated resource device handle will be stored
resourcetype	type of resource. Possible values are: • GC_VOICEDEVICE • GC_MEDIADEVICE • GC_NET_GCLINEDEVICE (only valid if the **linedev** parameter is an IP media 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>
#include <gcip.h>
#include <gcip_defs.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,
                 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

None

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: gclib.h
gcerr.h

Category: system controls and tools

Mode: synchronous

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

Termination 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 <gcip.h>
#include <gcip_defs.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%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);
    }

    *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, releasenump, intnump, component)

Inputs:

LINEDEV linedev	• Global Call line device handle
unsigned int *releasenump	• pointer to location where production release number will be stored
unsigned int *intnump	• pointer to location where internal release number will be stored
long component	• system component

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: optional call handling

Mode: synchronous

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.

A 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

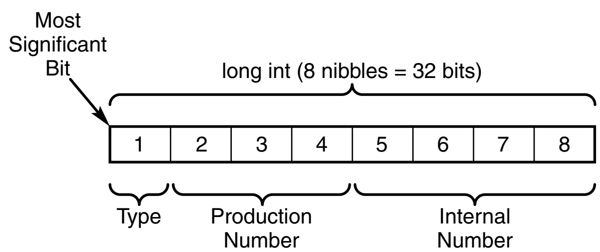
For example:

- 1.00 Production
- 1.00 Beta 5

Parameter	Description
linedev	Global Call line device handle. If this parameter is set to 0, the version number of the Global Call API is returned.
releasenump	points to the location where the production release number and type indicator will be stored
intnump	points to the location where the internal release number will be stored
component	specifies the software component to which the version number applies. For HMP, the only selection is: • GC_LIB – Global Call library

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

- *releasenump = 0x0102
- *intnump = 0x0000

For a version number of 1.02 beta 2, then:

- *releasenump = 0x1102
- *intnump = 0x0200

Table 5. gc_GetVer( ) Return Values

*releasenump			*intnump	
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>
#include <gcip.h>
#include <gcip_defs.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_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

■ 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.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
linedev	Global Call line device handle
sctsinfop	points to the SC_TSINFO CT Bus time slot information structure. See SC_TSINFO, on page 275 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>
#include <gcip.h>
#include <gcip_defs.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_Listen( )

- **gc_UnListen( )**

gc_InitXfer( )

Name: int gc_InitXfer(crn, parmblkp, ret_rerouting_infopp, mode)

Inputs:

CRN crn	• call reference number for the call between transferring party A and remote transferred-to party C
GC_PARM_BLK *parmblkp	• pointer to the parameter block associated with this request
GC_REROUTING_INFO ** ret_rerouting_infopp	• pointer to the location that stores the pointer to the returned rerouting information generated by the call control library
unsigned long mode	• async

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: supplementary service - call transfer

Mode: asynchronous

■ Description

The **gc_InitXfer( )** function is called by the transferring party (party A) to the remote transferred-to party (party C) to initiate a supervised transfer. With some technologies and protocols, party A is required to call this function prior to invoking a call transfer. The function is used to determine whether the remote party C can participate in the call transfer and to obtain calling information for the transfer.

After receiving the unsolicited event GCEV_REQ_INIT_XFER, the remote party C can accept the initiate transfer request by calling **gc_AcceptInitXfer( )** or can reject the initiate transfer request by calling **gc_RejectInitXfer( )**. In return, party A is notified of the acceptance or rejection from remote party C, which may be associated with rerouting information.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
crn	call reference number for the call between the transferring party A and the remote transferred-to party C receiving the initiate transfer request
parmblkp	points to the GC_PARM_BLK structure. The structure lists the data needed for requesting an initiate transfer. This parameter is optional and should be set to 0 if not used.
ret_rerouting_ infopp	not used in IP; should be set to NULL
mode	set to EV_ASYNC for asynchronous execution

After receiving GCEV_INIT_XFER to indicate that the initiate transfer request was successful, the application at party A can obtain the rerouting destination number or address from extevtdatap in the METAEVENT data structure associated with the GCEV_INIT_XFER event. This rerouting destination number or address can be used in the subsequent **gc_InvokeXfer( )** function that invokes the call transfer. The application does not need to allocate or deallocate the extevtdatap.

Termination Events

GCEV_INIT_XFER
: indicates that the **gc_InitXfer( )** function was successful (implies that the request was accepted by remote party C)

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

GCEV_INIT_XFER_REJ
: indicates that the initiate transfer request was successfully invoked by party A via the **gc_InitXfer( )** function, but the request was rejected by remote party C

Cautions

- For supervised call transfer, **gc_InitXfer( )** must be called before **gc_InvokeXfer( )**.
- The **gc_InitXfer( )** function can be called only when the call is in the GCST_CONNECTED call state.
- After receiving GCEV_INIT_XFER_FAIL or GCEV_INIT_XFER_REJ, the application should not call the **gc_InvokeXfer( )** function to invoke the call transfer.

Errors

If this function returns <0 to indicate failure, use the **gc_ErrorInfo( )** function for error information. If the GCEV_INIT_XFER_FAIL or GCEV_INIT_XFER_REJ 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).

Additional result values for GCEV_INIT_XFER_FAIL include:

GCRV_LOCALPARTY_PROT_TIMEOUT
: local party A protocol time-out

GCRV_REMOTEPARTY_PROT_TIMEOUT
: remote party C protocol time-out

Additional result values for GCEV_INIT_XFER_REJ include:

GCRV_REMOTEREJ_NOTALLOWED
: remote rejected; service not allowed

GCRV_REMOTEREJ_NOTSUBSCRIBED
: remote rejected; user is not subscribed

GCRV_REMOTEREJ_UNAVAIL
: remote rejected; service unavailable

GCRV_REMOTEREJ_UNSPECIFIED
: remote rejected; reason not given

Example

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

#define MAX_CHAN 30                /* maximum number of channels in system */
/*
* Data structure which stores all information for each line
*/
static struct linebag {
    LINEDEV    ldev;               /* Global Call API line device handle */
    CRN        crn;                /* Global Call API call handle */
    CRN        consultation_crn;   /* Global Call API call handle */
    int        blocked;            /* channel blocked/unblocked */
} port[MAX_CHAN+1];
struct linebag *pline;             /* pointer to access line device */

/*
 * Assume the following have been done:
 *    1. Opened line devices for each time slot on the network interface board.
 *    2. Each line device is stored in linebag structure "port".
 *    3. The consultation call has been established between Transferring party A and
 *       Transferred-To party C and the call is in connected or on hold state.
 */
int initiate_calltransfer(int port_num)
{
    GC_INFO       gc_error_info;        /* Global Call error information data */
    GC_PARM_BLK  * gc_pRetParmBlk = NULL; /* Returned the GC_PARM_BLK */

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

    /* Initiate the call transfer request */
    if (gc_InitXfer(pline->crn, NULL, &gc_pRetParmBlk, EV_ASYNC) == -1)
    {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_InitXfer() 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_AcceptInitXfer( )**
- **gc_AcceptXfer( )**
- **gc_InvokeXfer( )**

- **gc_RejectInitXfer( )**
- **gc_RejectXfer( )**

gc_InvokeXfer( )

Name: int gc_InvokeXfer(crn, extracrn, numberstr, makecallp, timeout, mode)

Inputs:

CRN crn	• call reference number for the call between transferring party A and the remote party receiving the call transfer request
CRN extracrn	• call reference number for the call between transferring party A and another remote party. Unused for blind call transfer.
char *numberstr	• rerouting number or address to transfer destination
GC_MAKECALL_BLK *makecallp	• pointer to the additional data for making the rerouting outbound call
int timeout	• time-out value
unsigned long mode	• async

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: supplementary service - call transfer

Mode: asynchronous

Description

The **gc_InvokeXfer( )** function is called by the transferring party (party A) to invoke a blind or supervised call transfer to the remote party that will make the rerouting outbound call. In a blind call transfer, party A provides the transferred party (party B) with the calling address of the transferred-to party (party C). In a supervised call transfer, the second call between party A and party C must also be established prior to this function call, and party A needs to specify party B or party C which will issue the third call and provide the calling address as well.

The remote party (B or C) that receives the call transfer request will be notified via an unsolicited event GCEV_REQ_XFER. The remote receiving party can accept the call transfer request by calling **gc_AcceptXfer( )** or can reject the call transfer request by calling **gc_RejectXfer( )**. Party A is always notified of the rejection but may or may not be notified of the acceptance, depending on the technology and protocol. For applications using H.323/H.450.2, the remote transferring party A will not receive notification of the acceptance. For applications using SIP, the remote transferring party A can optionally receive notification of the acceptance as a GCEV_INVOKE_XFER_ACCEPTED event.

Once the call transfer is completed (that is, party B and party C have established a call in Alerting or Connected state), party A receives GCEV_DISCONNECTED with a GCEV_XFERCALL_CMPLT result value for the primary call between A and B (and for the consultation call between A and C, if any). The application at party A then calls the **gc_DropCall( )** and **gc_ReleaseCallEx( )** functions to drop and release the call(s).

If the call transfer fails or is rejected, party A is notified and returns to its original call state.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
crn	call reference number for the call between the transferring party A and the remote party receiving the call transfer request. The call is the primary call in blind call transfer, and can be either the primary or secondary call in supervised call transfer depending on which party is required to make the rerouting call.
extracrn	call reference number for the call between the transferring party A and another remote party, e.g., the consultation call for a supervised transfer. For blind call transfer, this parameter is set to 0 when not used. In supervised call transfer, this field is mandatory and can represent either the primary or secondary call, depending on which call is represented by **crn**.
numberstr	rerouting number or address used by the remote receiving party to make an outbound call. The number must be terminated with '\0'.
makecallp	points to the GC_MAKECALL_BLK structure. The structure lists the parameters used to make the rerouting outbound call. The usage of GC_MAKECALL_BLK in **gc_InvokeXfer( )** is similar to that in **gc_MakeCall( )** except that the source address may be ignored.
timeout	ignored for IP
mode	set to EV_ASYNC for asynchronous execution

The extra CRN parameter is provided in **gc_InvokeXfer( )** because the second call at party A must be explicitly specified in a supervised transfer. Two input fields are supplied for calling address, to be consistent with the **gc_MakeCall( )** function.

■ Termination Events

GCEV_INVOKE_XFER
: indicates that the call transfer was successfully completed by the **gc_InvokeXfer( )** function

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

GCEV_INVOKE_XFER_REJ
: indicates that the call transfer request was successfully invoked by party A via the **gc_InvokeXfer( )** function, but the request was rejected by the remote party that received the call transfer request

■ Cautions

The **gc_InvokeXfer( )** function can be called only when the call between party A and remote receiving party is in the GCST_CONNECTED call state and another call, if any, is also in the GCST_CONNECTED call state.

Errors

If this function returns <0 to indicate failure, use the **gc_ErrorInfo( )** function for error information. If the GCEV_INVOKE_XFER_FAIL or GCEV_INVOKE_XFER_REJ 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>
#include <gcip.h>
#include <gcip_defs.h>
#include <gccfgparm.h>

#define MAX_CHAN 30                /* maximum number of channels in system */
/*
* Data structure which stores all information for each line
*/
static struct linebag {
    LINEDEV    ldev;               /* Global Call API line device handle */
    CRN        crn;                /* Global Call API call handle */
    CRN        consultation_crn;   /* Global Call API call handle */
    int        blocked;            /* channel blocked/unblocked */
} port[MAX_CHAN+1];
struct linebag *pline, * extra_pline;              /* pointer to access line device */
/*
 * Assume the following have been done:
 *    1. Opened line devices for each time slot on the network interface board.
 *    2. Each line device is stored in linebag structure "port".
 *    3. The original (primary) call has been established between Transferring party A and
 *       Transferred party B and the call is in connected or on holdanswered state.
 *    4. For blind (unsupervised) call transfer, the consultation_crn must be set to 0.
 *    5. For supervised call transfer, a call has been established between Transferring Party A
 *       and Transferred-To Party C and the call is in connected or on hold state and associated
 *       with the line device index extra_port_num.  Dependent on the technology/protocol, the
 *       consultation call and the original call may or may not share the same line device.
 */

int invoke_calltransfer(int port_num, int extra_port_num)
{
    GC_INFO   gc_error_info;    /* Global Call error information data */
    CRN       extra_crn = 0;    /* Extra Global Call API call handle for supervised transfer */

    /* Find info for this time slot, specified by 'port_num' */
    /* (assumes port_num is valid) */
    pline = port + port_num;
    extra_pline = NULL;
    if (port_num == extra_port_num)
    {
        /* If the consultation call and the original call associate the same line device,
         * assign the consultation crn as the extra_crn */
        extra_crn = port->consultation_crn;
    }
    else
    {
        /* If the consultation call and the original call do not associate the same line device,
```

```
         * find the line device of the consultation call (assumes extra_port_num is valid)
         * and assign the crn as the extra_crn */
        extra_pline = port + port_num;
        extra_crn = extra_pline->crn;
    }

    /* Reroute the original call to the destination at 127.0.0.1 */
    if (gc_InvokeXfer(pline->crn, 0, "TA:127.0.0.1", NULL, 0, EV_ASYNC) == -1)
    {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_InvokeXfer() 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_AcceptInitXfer( )**
- **gc_AcceptXfer( )**
- **gc_InitXfer( )**
- **gc_RejectInitXfer( )**
- **gc_RejectXfer( )**

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

■ Description

The **gc_LinedevToCCLIBID( )** function returns the call control library ID of the call control library that opened the specified line device.

Parameter	Description
linedev	Global Call line device handle
cclibid	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>
#include <gcip.h>
#include <gcip_defs.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 cclib 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

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.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
linedev	Global Call line device handle
sctsinfop	points to the SC_TSINFO CT Bus time slot information structure. See SC_TSINFO, on page 275 for more information.
mode	set to EV_ASYNC for asynchronous execution or to EV_SYNC 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>
#include <gcip.h>
#include <gcip_defs.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_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

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: basic

Mode: asynchronous

■ 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.

Note: Use of the **gc_MakeCall( )** function is technology-specific. See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
linedev	Global Call line device handle
crnp	points to the memory location where the call reference number is to be stored
numberstr	called party's telephone number (must be terminated with "\0"). Maximum length is 32 digits.
makecallp	points to the GC_MAKECALL_BLK structure. The GC_MAKECALL_BLK structure lists the parameters used to make an outbound call. Assigning a NULL to the **makecallp** parameter indicates that the default values should be used for the call.

Parameter	Description
timeout	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.
mode	set to EV_ASYNC for asynchronous execution

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

■ 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 **usrattr** parameter points to a buffer where a user defined attribute is stored. 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.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
linedevp	points to unique number to be filled in by this function to identify a specific device
devicename	points to an ASCII string that defines the device(s) associated with the returned **linedevp** number. The **devicename** parameter specifies the device to be opened and the protocol to be used. The format used to define **devicename** is: <field1><field2>...<fieldn> These fields may be listed in any order. The field format is: :<key>_<field name> Valid keys and their appropriate field names are: • **P** - *protocol_name* – specifies the protocol to be used. See the *Global Call IP for Host Media Processing Technology Guide* for technology specific protocol information. • **N** - *network_device_name* – specifies the board name and the time slot name (if needed) using the following naming convention: If the board is to be opened, the *network_device_name* is the board name. An example of the format is: **iptB<number of board>** If a time slot is to be opened, both the board and time slot are specified. An example of the format is: **iptB<number of board>T<time slot number>** See the *Global Call IP for Host Media Processing Technology Guide* for the correct *network_device_name*. • **V** - *voice_device_name* – specifies the voice board and channel. An example of the format is: **dxxxB<virtual board number>C<channel number>** • **M** - *media_device_name* – specifies the media board and channel. An example of the format is: **ipmB<virtual board number>C<channel number>** Not all technologies support voice or media devices. See the *Global Call IP for Host Media Processing Technology Guide* for the correct voice or media device names, if supported.
mode	set to EV_ASYNC for asynchronous execution or to EV_SYNC for synchronous execution
usrattr	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 IP for Host Media Processing Technology Guide* 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.
- 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.

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 <dxxxlib.h>
#include <dtilib.h>

/*
 * GlobalCall header(s)
 */
#include <gclib.h>
#include <gcerr.h>
#include <gcip.h>
#include <gcip_defs.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_enbhdlr(EV_ANYDEV, EV_ANYEVT,
        (long (*) (void *))evt_hdlr) == -1) {
        printf("sr_enbhdlr failed\n");
        exit(1);
    }
```

```
    /* Start the library */
    if (gc_Start(NULL) != GC_SUCCESS) {
        /* process error return as shown */
        print_error("gc_Start");
    }

    /* open the line devices */
    open_line_devices();

    sr_waitevt(50);

    /* close the line devices */
    close_line_devices();

    /* Stop the library */
    if (gc_Stop() != GC_SUCCESS) {
        /* process error return as shown */
        print_error("gc_Stop");
    }
}

/*
 * int print_error (char *function)
 *
 * INPUT: char *function - function name
 * RETURN: gc_error      - globalcall error number
 *
 */
int print_error(char *function)
{
    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;                        /* reason for failure of function */
    METAEVENT         metaevent;

    if (gc_GetMetaEvent(&metaevent) != GC_SUCCESS) {
        /* process error return as shown */
        error = print_error("gc_GetMetaEvent");
        return(error);
    }

    if (metaevent.flags & GCME_GC_EVENT) {
        /* process GlobalCall events */
```

```
        if (gc_GetUsrAttr(metaevent.linedev, (void **)&pline) != GC_SUCCESS) {
            /* process error return as shown */
            error = print_error("gc_GetUsrAttr");
            return(error);
        }

        switch (metaevent.evttype) {
           case GCEV_UNBLOCKED:
              printf("received GCEV_UNBLOCKED event on %s\n", ATDV_NAMEP(pline->networkh));
              pline->blocked = 0;
           break;

           default:
              printf ("Unexpected GlobalCall event received\n");
           break;
        }
    }
    else {
        /* process other events */
    }

       return 0;
}

/*
 * int open_line_devices (void)
 *
 * RETURN: 0             - function successful
 *         error         - GlobalCall error number
 *
 */
int open_line_devices(void)
{
    char            devname[64];          /* argument to gc_OpenEx() function */
    int             vbnum = 0;            /* virtual board number (1-based) */
    int             vch = 0;              /* voice channel number (1-based) */
    int             ts;                   /* time slot number (1-based) */
    int             port_index;           /* index for 'port' */
    int             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++) {

        vbnum = (ts - 1) / 4 + 1;
        vch = ((ts - 1) % 4) + 1;
        sprintf (devname, ":N_iptB1T1:P_IP:M_ipmB1C1:V_dxxxB1C1", ts, vbnum, 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_MEDIADEVICE)) != GC_SUCCESS) {
            /* process error return as shown */
            error = print_error("gc_GetResourceH(GC_MEDIADEVICE)");
            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_GetResourceH");
            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_RejectInitXfer( )

Name:	int gc_RejectInitXfer(crn, reason, parmblkp, mode)	
Inputs:	CRN crn	• call reference number for the call between remote transferring party A and transferred-to party C
	unsigned long reason	• reason for rejection
	GC_PARM_BLK *parmblkp	• pointer to the parameter block associated with the rejection
	unsigned long mode	• async
Returns:	0 if successful <0 if failure	
Includes:	gclib.h gcerr.h	
Category:	supplementary service - call transfer	
Mode:	asynchronous	

■ Description

The **gc_RejectInitXfer( )** function is called by the transferred-to party (party C) to reject the initiate transfer request from the remote transferring party (party A), indicating that party C cannot participate in the call transfer. Prior to calling this function, party C had to be notified of the initiate transfer request via an unsolicited event GCEV_REQ_INIT_XFER.

The remote transferring party A is notified of the rejection via a GCEV_INIT_XFER_REJ event.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
crn	call reference number for the call between the remote transferring party A and the transferred-to party C that received the initiate transfer request
reason	specifies the value of rejection reason, which may be: • GCVAL_REJREASON_NOTALLOWED • GCVAL_REJREASON_NOTSUBSCRIBED • GCVAL_REJREASON_UNAVAIL • GCVAL_REJREASON_UNSPECIFIED
parmblkp	points to the GC_PARM_BLK structure. The structure lists the data needed for rejection. This parameter is optional and should be set to 0 if not used. Note that party C is not required to provide its rerouting address via the **parmblkp** parameter (as when using the **gc_AcceptInitXfer( )** function).
mode	set to EV_ASYNC for asynchronous execution

Regardless of whether the **gc_RejectInitXfer( )** function succeeds or fails, the call state of party C returns to its original call state (GCST_CONNECTED).

Preceding Events

GCEV_REQ_INIT_XFER
: unsolicited event, notifies application at party C of the initiate transfer request from remote party A

Termination Events

GCEV_REJ_INIT_XFER
: indicates that the **gc_RejectInitXfer( )** function was successful, that is, party C successfully rejected the initiate transfer request from remote party A

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

Cautions

The **gc_RejectInitXfer( )** function can be called only when the call is in the GCST_REQ_INIT_XFER call state.

Errors

If this function returns <0 to indicate failure, use the **gc_ErrorInfo( )** function for error information. If the GCEV_REJ_INIT_XFER_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>
#include <gcip.h>
#include <gcip_defs.h>

#define MAX_CHAN 30                /* maximum number of channels in system */
/*
* Data structure which stores all information for each line
*/
static struct linebag {
    LINEDEV    ldev;               /* Global Call API line device handle */
    CRN        crn;                /* Global Call API call handle */
    CRN        consultation_crn;   /* Global Call API call handle */
    int        blocked;            /* channel blocked/unblocked */
} port[MAX_CHAN+1];
struct linebag *pline;             /* pointer to access line device */
/*
 * Assume the following have been done:
 *    1. Opened line devices for each time slot on the network interface board.
 *    2. Each line device is stored in linebag structure "port".
```

```
 *    3. The original (primary) call has been established between Transferring party A and
 *       Transferred party B and the call is in connected or on hold state.
 *    4. The party C received the GCEV_REQ_INIT_XFER.
 */
int reject_transferinitiate(int port_num, unsigned long reason)
{
    GC_INFO        gc_error_info;    /* Global Call error information data */

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

    /* Reject the call transfer request */
    if (gc_RejectInitXfer(pline->crn, reason, NULL, EV_ASYNC) == -1)
    {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_RejectInitXfer() 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_AcceptInitXfer( )**
- **gc_AcceptXfer( )**
- **gc_InitXfer( )**
- **gc_InvokeXfer( )**
- **gc_RejectXfer( )**

gc_RejectXfer( )

Name: int gc_RejectXfer(crn, reason, parmblkp, mode)

Inputs:

CRN crn	• call reference number for the call between remote transferring party A and the local party receiving the call transfer request
unsigned long reason	• reason for rejection
GC_PARM_BLK *parmblkp	• pointer to the parameter block associated with the rejection
unsigned long mode	• async

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: supplementary service - call transfer

Mode: asynchronous

Description

The **gc_RejectXfer( )** function is called by the local party (transferred party B) to reject the call transfer request from the remote transferring party A. Prior to calling this function, the local party had to be notified of the call transfer request via an unsolicited event GCEV_REQ_XFER, which associated the information for the rerouting number or address.

The remote transferring party A is notified of the rejection via a GCEV_INVOKE_XFER_REJ event.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
crn	call reference number for the call between the remote transferring party A and the local party that received the call transfer request
reason	specifies the value of rejection reason, which may be protocol-specific. For H.323/H.450.2, the defined rejection reasons are: • GCVAL_REJREASON_INVADDR • GCVAL_REJREASON_INSUFFINFO • GCVAL_REJREASON_NOTALLOWED • GCVAL_REJREASON_NOTSUBSCRIBED • GCVAL_REJREASON_UNAVAIL • GCVAL_REJREASON_UNSPECIFIED This parameter is mandatory.

Parameter	Description
parmblkp	ignored for IP
mode	set to EV_ASYNC for asynchronous execution

After successfully rejecting the request, the call state of local party B or C returns to its original call state (GCST_CONNECTED).

When receiving the GCEV_REQ_XFER event, the application can obtain the information for the rerouting number or address from extevtdatap in the METAEVENT data structure. Because the extevtdtatap is managed by GCLib/CCLib (allocated/deleted automatically), the user application does not need to allocate or deallocate extevtdatap.

■ Preceding Events

GCEV_REQ_XFER
: unsolicited event, notifies application at the local party receiving the call transfer request from remote party A

■ Termination Events

GCEV_REJ_XFER
: indicates that the **gc_RejectXfer( )** function was successful, that is, the local party successfully rejected the call transfer request from the remote party

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

■ Cautions

The **gc_RejectXfer( )** function can be called only when the call to be transferred is in the GCST_REQ_XFER call state (that is, after receiving GCEV_REQ_XFER).

■ Errors

If this function returns <0 to indicate failure, use the **gc_ErrorInfo( )** function for error information. If the GCEV_REJ_XFER_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>
#include <gcip.h>
#include <gcip_defs.h>
#include <gccfgparm.h>
```

```
#define MAX_CHAN 30                /* maximum number of channels in system */
/*
* Data structure which stores all information for each line
*/
static struct linebag {
    LINEDEV    ldev;               /* Global Call API line device handle */
    CRN        crn;                /* Global Call API call handle */
    CRN        consultation_crn;   /* Global Call API call handle */
    int        blocked;            /* channel blocked/unblocked */
} port[MAX_CHAN+1];
struct linebag *pline;             /* pointer to access line device */
/*
 * Assume the following have been done:
 *    1. Opened line devices for each time slot on the network interface board.
 *    2. Each line device is stored in linebag structure "port".
 *    3. The original (primary) call has been established between Transferring party A and
 *       Transferred Party B and the call is in connected or on holdanswered state.
 *    4. The party B received the GCEV_REQ_XFER.
 */
int reject_calltransfer(int port_num, unsigned long reason)
{
    GC_INFO        gc_error_info;    /* Global Call error information data */

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

    /* Reject the call transfer request */
    if (gc_RejectXfer(pline->crn, reason, NULL, EV_ASYNC) == -1)
    {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_RejectXfer() 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_AcceptInitXfer( )**
- **gc_AcceptXfer( )**
- **gc_InitXfer( )**
- **gc_InvokeXfer( )**
- **gc_RejectInitXfer( )**

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

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( )**.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
crn	call reference number for the call
mode	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>
#include <gcip.h>
#include <gcip_defs.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_ReqService( )

Name: int gc_ReqService(target_type, target_ID, pserviceID, reqdatap, respdatapp, 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_BLKP reqdatap	• pointer to data associated with the request
GC_PARM_BLKP *respdatapp	• pointer to location that stores the pointer to the response buffer generated by the call control library
unsigned long mode	• async

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: Global Call Service Request (GCSR)

Mode: asynchronous

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.

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*.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
target_type	target object type. Valid values are: • GCTGT_GCLIB_CHAN • GCTGT_GCLIB_CRN
target_id	target object identifier. This identifier, along with **target_type**, uniquely specifies the target object. Valid identifiers are: • line device handle • call reference number
pserviceID	points to the service ID of this request. Assigned by the call control library when this function returns.
reqdatap	points to the user-specified data associated with the request

Parameter	Description
respdatapp	set to NULL in asynchronous mode
mode	set to EV_ASYNC for asynchronous 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 *Global Call IP for Host Media Processing Technology Guide* for technology-specific information.

Cautions

- Only synchronous mode is supported for the following target object: GCTGT_GCLIB_SYSTEM. Otherwise, the function will return the async mode error.
- 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 *isdnerr.h* file for the ISDN call control library).

Example

```
#include "gclib.h"
#include "gcip.h"
#include "gcip_defs.h"

void main( )
{
    LINEDEV          devh;
    unsigned long    serviceID;
    GC_PARM_BLKP     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

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: system controls and tools

Mode: asynchronous

■ 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.

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.

Parameter	Description
linedev	Global Call line device
mode	set to EV_ASYNC for asynchronous 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.

■ 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 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 <gclib.h>
#include <gcerr.h>
#include <gcip.h>
#include <gcip_defs.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_ASYNC) != 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_BLKP datap	• pointer to data associated with the response
unsigned long mode	• async

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: Global Call Service Request (GCSR)

Mode: asynchronous

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*.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
target_type	target object type. Valid values are: • GCTGT_GCLIB_CHAN • GCTGT_GCLIB_CRN
target_id	target object identifier. This identifier, along with **target_type**, uniquely specifies the target object. Valid identifiers are: • line device handle • call reference number
datap	points to data associated with the response; contains user-specified values
mode	set to EV_ASYNC for asynchronous 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 *Global Call IP for Host Media Processing Technology Guide* for technology-specific information.

■ Cautions

- 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"
#include "gcip.h"
#include "gcip_defs.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_ASYNC ) != 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

Description

The **gc_ResultInfo( )** function retrieves information about Global Call events. 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
a_Metaevent	points to the metaevent associated with this event
a_Info	points to the GC_INFO structure where information about the event is contained. See GC_INFO, on page 258 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_CCLibIDToName( )**

gc_SetAlarmConfiguration( )

Name: int gc_SetAlarmConfiguration(linedev, aso_id, alarm_list, alarm_config_type)

Inputs:

LINEDEV linedev	• Global Call line device handle
unsigned long aso_id	• alarm source object ID
ALARM_LIST *alarm_list	• pointer to alarm list
int alarm_config_type	• alarm information type

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: GCAMS

Mode: synchronous

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.

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 244.) 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 243.) 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
linedev	Global Call line device handle. The **linedev** 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.
aso_id	alarm source object (ASO) ID. Use the **gc_AlarmSourceObjectNameToID( )** 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.
alarm_list	points to the alarm list. The alarm list will contain the list of alarms to be passed to the application. (See ALARM_LIST, on page 244.) NULL is not allowed.
alarm_config_type	performs two roles. The first role is to specify whether the application wants to be notified when any of the alarms in the **alarm_list** occur and to classify the alarms in the **alarm_list** as either blocking or non-blocking. Possible values are: • ALARM_CONFIG_BLOCKING – indicates whether the specified alarms for the given **aso_id** and **linedev** are to be blocking or non-blocking. The data field for each alarm in **alarm_list** will contain either ALARM_BLOCKING (for blocking alarms) or ALARM_NONBLOCKING (for nonblocking alarms). • ALARM_CONFIG_NOTIFY – indicates whether the application wants to be notified of the alarms in the **alarm_list** for the given **aso_id** and **linedev**. The data field for each alarm in **alarm_list** will contain either ALARM_NONOTIFY (if the application is not to be notified) or ALARM_NOTIFY (if the application is to be notified). 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: • ALARM_CONFIGURE_BOARDS – configure all the boards for the specified ASO ID • ALARM_CONFIGURE_BOARDS_AND_TIMESLOTS – configure all the boards and time slots for the specified ASO ID • ALARM_CONFIGURE_TIMESLOTS – configure the time slots for the specified board line device See Table 6 for more information about setting flags.

Table 6. Possible Scope Settings for the alarm_config_type Parameter

Object(s) to Configure: ASO	Board	Time Slot	alarm_config_type Flag Setting†	linedev Setting
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 **linedev** ID
no	yes	no	none	Board **linedev** ID
no	no	yes	none	Time slot **linedev** 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 **alarm_config_type** 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_NONOTIFY))
            {
                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_SetAlarmFlow( )**
- **gc_SetAlarmNotifyAll( )**

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

■ 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.

Parameter	Description
aso_id	alarm source object (ASO) ID. Use the **gc_AlarmSourceObjectNameToID( )** function to obtain the ASO ID for the desired alarm source object.

Parameter	Description
flow	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: • ALARM_FLOW_ALWAYS – all alarms with notification on are sent to the application (default) • ALARM_FLOW_ALWAYS_BLOCKING – all blocking alarms with notification on are sent to the application. Non-blocking alarms are not sent. • 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. • 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)

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>
#include <gcip.h>
#include <gcip_defs.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	• Global Call line device handle
	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	

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
linedev	Global Call line device handle. The **linedev** parameter must be set to NULL to set the notification attribute for alarm source objects. Setting **linedev** to NULL changes the default behavior for any device opened after the function call.

Parameter	Description
aso_id	alarm source object (ASO) ID. Use the **gc_AlarmSourceObjectNameToID( )** 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.
value	performs two roles. The first role is to set the alarm notification attribute. Possible values are: • ALARM_NOTIFY – notify the application of all alarms associated with the **aso_id** • ALARM_NONOTIFY – do not notify the application of alarms associated with the **aso_id** 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: • ALARM_CONFIGURE_BOARDS – configure all the boards for the specified ASO ID • ALARM_CONFIGURE_BOARDS_AND_TIMESLOTS – configure all the boards and time slots for the specified ASO ID • ALARM_CONFIGURE_TIMESLOTS – configure the time slots for the specified board See Table 7 for more information about setting these flags. Only one of the three flags may be set at a time.

Table 7. Possible Flag Settings for value Parameter

Object(s) to Configure: ASO	Board	Time Slot	alarm_config_type Flag Setting†	linedev Setting
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 **linedev** ID
no	yes	no	none	Board **linedev** ID
no	no	yes	none	Time slot **linedev** 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 *isdnerr.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:

LINEDEV linedev	• Global Call line device handle
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

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).

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.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
linedev	Global Call line device handle
aso_id	alarm source object (ASO) ID. Use the **gc_AlarmSourceObjectNameToID( )** 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.
ParmSetID	identifies a set of associated parameters

Parameter	Description
alarm_parm_list	points to the alarm parameter list. See ALARM_PARM_LIST, on page 246 for a description of the fields in the ALARM_PARM_LIST data structure.
mode	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 *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 */
#include <gcip.h>
#include <gcip_defs.h>

int main(int argc, char **argc)
{
    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 signficance */
    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_SetAuthenticationInfo( )

Name: int gc_SetAuthenticationInfo(target_type, target_id, infoparmblkp)

Inputs:

int target_type	• type of target object (virtual board)
long target_id	• target object ID
GC_PARM_BLKP infoparmblkp	• pointer to GC_PARM_BLK with user information

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: FTE

Mode: synchronous

Description

The **gc_SetAuthenticationInfo( )** function is used to configure or remove authentication information on an IPT virtual board device.

This function is specific to the IP technology, and is documented in detail in the *Global Call IP for Host Media Processing Technology Guide*.

gc_SetConfigData( )

Name: int gc_SetConfigData(target_type, target_id, target_datap, time_out, update_cond, request_idp, mode)

Inputs:

int target_type	• target object type
long target_id	• target object ID
GC_PARM_BLKP 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

Returns: 0 if successful
<0 if failure

Includes: gcerr.h

Category: RTCM, system controls and tools

Mode: asynchronous

Description

The **gc_SetConfigData( )** function supports the Global Call Real Time Configuration Management (RTCM) 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 14, "Global Call Parameter Entry List Maintained in GCLIB", on page 291 and Table 15, "Examples of Parameter Entry List Maintained in CCLIB", on page 293. For more information about the RTCM feature, see the *Global Call API Programming Guide*.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
target_type	target object type
target_id	target object identifier. This identifier, along with **target_type**, uniquely specifies the target object.
target_datap	points to the GC_PARM_BLK structure. This structure contains the parameter configuration data to be updated. See GC_PARM_BLK, on page 261 for more information.
time_out	not supported in asynchronous mode

Parameter	Description
update_cond	specifies when to update. Valid values are: • GCUPDATE_IMMEDIATE – requires parameter to be updated immediately • GCUPDATE_ATNULL – requires parameter to be updated at the Null call state. Applicable only if the **target_type** is associated with a call.
request_idp	points to the location for storing the request ID, which is generated by Global Call
mode	set to EV_ASYNC for asynchronous 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 261 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 8.

Table 8. 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 **gc_SetConfigData( )** with GCUPDATE_ATNULL and then call **gc_ResetLineDev( )**.

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 266 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.) 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 *Global Call IP for Host Media Processing Technology Guide* for the impact of changing any settings.
- Only synchronous mode is supported for the following target object: GCTGT_GCLIB_SYSTEM. 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>
#include <gcip.h>
#include <gcip_defs.h>

    #define MAXCHAN 30
/*
    Data structure that stores all information about each line device
*/
struct linebag
{
    LINEDEV   ldev;          /* Line device ID */
    CRN       crn;           /* CRN */
    int       call_sate;
} port[MAXCHAN];
```

```
    struct linebag     *pline;

    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;
    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;
    gc_util_insert_parm_val(&parm_data_blkp, GCSET_CALLSTATE_MSK, GCACT_SETMSK,
                            sizeof(long), long_value);

    /* 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");
        }
    }

    /* delete the parm data block after using it */
    gc_util_delete_parm_blk(parm_data_blkp);
    return 0;
}
```

■ See Also

None

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_BLKP 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

■ 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.

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 287 for more information.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
target_type	target object type. Valid values are: • GCTGT_GCLIB_CHAN • GCTGT_GCLIB_CRN See Table 14, "Global Call Parameter Entry List Maintained in GCLIB", on page 291 for details.
target_id	target object identifier. This identifier, along with target_type, uniquely specifies the target object.
infoparmblkp	points to GC_PARM_BLK which contains parameters defining user information. See the *Global Call IP for Host Media Processing Technology Guide* for GC_PARM_BLK pointer referenced structure definitions.

Parameter	Description
duration	duration of the user information block setting: • 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. • GC_ALLCALLS – effective for the current or next call, and all subsequent calls.

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

The *Global Call IP for Host Media Processing Technology Guide* contains technology-specific examples of using the **gc_SetUserInfo( )** function.

See Also

None

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

■ 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
linedev	Global Call line device handle
usrattr	user defined attribute. Applications can recall this value by calling **gc_GetUsrAttr( )**.

■ 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 <gcip.h>
#include <gcip_defs.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_SipAck( )

Name: int gc_SipAck(crn, parmblk, mode)

Inputs:

CRN crn	• call reference number of call targeted for modification
GC_PARM_BLKP parmblk	• pointer to optional parameter block containing SDP content for the SIP ACK message
unsigned long mode	• completion mode (EV_ASYNC)

Returns: 0 if successful
<0 if unsuccessful

Includes: gclib.h

Category: third-party call control

Mode: asynchronous

■ Description

This SIP protocol specific function is used in third-party call control (3PCC) mode to send an explicit SIP ACK message to the remote party on an outbound INVITE or re-INVITE transaction when the library does not automatically send an ACK.

This function is specific to the IP technology, and is documented in detail in the *Global Call IP for Host Media Processing Technology Guide*.

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

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.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
startp	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.
- Applications intending to use Global Call over IP should ensure that the network adapter is enabled before calling **gc_Start( )**. If the application doesn't intend to use Global Call over IP and needs to keep network adapter disabled, the application should load the call control libraries selectively using the **GC_START_STRUCT** parameter of **gc_Start( )** and avoid loading GC_H3R_LIB and GC_IPM_LIB.

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 <gclib.h>
#include <gcerr.h>
#include <gcip.h>
#include <gcip_defs.h>

int sysinit()
{
    GC_START_STRUCT gclib_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_ICAPI_LIB",NULL},
        {"GC_PDKRT_LIB",NULL},
        {"GC_CUSTOM1_LIB",(void *)&custom_lib1},
       };

    gclib_start.num_cclibs = 4;
    gclib_start.cclib_list = cclib_start;
```

```
    /* Next issue a gc_Start() Call */
    if ( gc_Start( &gclib_start ) != GC_SUCCESS )    {
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_Start(), 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);
    }
    return (0);

    /* Next open the GlobalCall Line Devices */

return(0);
}
```

See Also

- **gc_CCLibStatusEx( )**
- **gc_Stop( )**

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

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>
#include <gcip.h>
#include <gcip_defs.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_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

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.

Note: See the *Global Call IP for Host Media Processing Technology Guide* for additional information about this function.

Parameter	Description
linedev	Global Call line device handle
mode	set to EV_ASYNC for asynchronous execution or to EV_SYNC 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>
#include <gcip.h>
#include <gcip_defs.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_copy_parm_blk( )

Name: int gc_util_copy_parm_blk(parm_blkpp, parm_blkp)

Inputs: GC_PARM_BLKP* parm_blkpp • pointer to the address of the new GC_PARM_BLK

GC_PARM_BLKP parm_blkp • pointer to a valid GC_PARM_BLK to be copied

Returns: GC_SUCCESS if successful
GC_FAIL if unsuccessful

Includes: gclib.h
gcerr.h

Category: GC_PARM_BLK utility

Mode: synchronous

Description

The **gc_util_copy_parm_blk( )** function copies the specified GC_PARM_BLK.

This function **must** be used to copy any GC_PARM_BLK that contains any setID/parmID pairs that can have data that is potentially larger than 255 bytes. This function can be used for any GC_PARM_BLK, regardless of whether it contains setID/parmID pairs that support parameter data lengths greater than 255 bytes.

Note: The only Global Call parameters that currently support data longer than 255 bytes are IPSET_SIP_MSGINFO/IPPARM_SIP_HDR, which is used for SIP message headers, and the parameters in the IPSET_SDP set, which is used in 3PCC mode.

Parameter	Description
parm_blkpp	pointer to the address of the new GC_PARM_BLK that the specified parm block will be copied to; **must** be set to NULL
parm_blkp	points to a valid, existing GC_PARM_BLK to be copied

Cautions

To avoid a memory leak, any GC_PARM_BLK created must eventually be deleted using the **gc_util_delete_parm_blk( )** function.

Errors

If this function returns GC_ERROR(-1) 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"
#include "gcip.h"
```

```
void process_event(void)
{
   METAEVENT  metaevent;
   GC_PARM_BLKP my_blkp = NULL;

   if(gc_GetMetaEvent(&metaevent) != GC_SUCCESS)
   {
      /* process error */
   }

   Switch(metaevent.evttype)
   {
      case GCEV_OFFERED:
         /* make a copy of the parm blk */
         if(metaevent.extevtdatap)
         {
            if ( gc_util_copy_parm_blk( &my_blkp,(GC_PARM_BLKP)(metaevent.extevtdatap))
                 != GC_SUCCESS )
            {
               /* Process error */
            }
         }
      ......
   }
   ......
}
```

See Also

- gc_util_delete_parm_blk( )

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: none

Includes: gclib.h
gcerr.h

Category: GC_PARM_BLK utility

Mode: synchronous

Description

The **gc_util_delete_parm_blk( )** function deletes the specified GC_PARM_BLK.

Parameter	Description
parm_blk	points to the GC_PARM_BLK to be deleted. For more information about the GC_PARM_BLK data structure, see GC_PARM_BLK, on page 261.

Termination Events

None

Cautions

None

Errors

None

Example

```
#include "gclib.h"
#include "gcip.h"
#include "gcip_defs.h"

void main( )
{
    GC_PARM_BLKP 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

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
parm_blk	points to a valid GC_PARM_BLK where the parameter is possibly located
setID	set ID of the parameter to be found
parmID	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

None

Example

```
#include "gclib.h"
#include "gcip.h"
#include "gcip_defs.h"

void main( )
{
    GC_PARM_BLKP 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_delete_parm_blk( )**
- **gc_util_insert_parm_ref( )**
- **gc_util_insert_parm_val( )**
- **gc_util_next_parm( )**

gc_util_find_parm_ex( )

Name:	int gc_util_find_parm_ex(parm_blk, setID, parmID, parm)	
Inputs:	GC_PARM_BLKP parm_blk	• pointer to GC_PARM_BLK to search for the parameter
	unsigned long setID	• parameter set ID of parameter to be found
	unsigned long parmID	• parameter ID of parameter to be found
	GC_PARM_DATA_EXTP parm	• pointer to a valid GC_PARM_DATA_EXT structure that identifies where in the parm block to start searching
Outputs:	GC_PARM_DATA_EXTP parm	• if successful, pointer to a GC_PARM_DATA_EXT structure that contains the ID and value data for the specified parameter
Returns:	GC_SUCCESS if successful EGC_NO_MORE_PARMS if no more parameters exist in GC_PARM_BLK GC_ERROR if failure	
Includes:	gclib.h gcerr.h	
Category:	GC_PARM_BLK utility	
Mode:	synchronous	

■ Description

The **gc_util_find_parm_ex( )** function is used to find a parameter of a particular type in a GC_PARM_BLK and retrieve the parameter data into a GC_PARM_DATA_EX structure.

This function **must** be used instead of the similar **gc_util_find_parm( )** function if the parameter data can potentially exceed 255 bytes. This function is backward compatible and can be used instead of **gc_util_find_parm( )** for any GC_PARM_BLK, regardless of whether the parameter block contains setID/parmID pairs that support data lengths greater than 255 bytes.

Note: The only Global Call parameters that currently support data longer than 255 bytes are IPSET_SIP_MSGINFO/IPPARM_SIP_HDR, which is used for SIP message headers, and the parameters in the IPSET_SDP set, which is used in 3PCC mode.

The **gc_util_find_parm_ex( )** function can be used to determine whether a particular parameter exists, or to retrieve a particular parameter, or both. If the specified parameter is found in the GC_PARM_BLK, the function fills in the GC_PARM_DATA_EX structure with the parameter data and returns GC_SUCCESS. If the parameter does not exist in the GC_PARM_BLK, or if no more parameters of the specified type are found, the function returns EGC_NO_MORE_PARMS.

To search from the beginning of the GC_PARM_BLK, initialize the GC_PARM_DATA_EXT structure (**parm**) by using **INIT_GC_PARM_DATA_EXT(parm)** before calling **gc_util_find_parm_ex( )**. If the structure pointed to by **parm** contains parameter information that was retrieved in a previous call to this function, the function will begin its search at that parameter rather than the beginning of the parameter block.

Parameter	Description
parm_blk	points to a valid GC_PARM_BLK that will be searched for a parameter of the specified type
setID	set ID of the parameter to be found
parmID	parameter ID of the parameter to be found
parm	points to a valid GC_PARM_DATA_EXT provided by the application. If a pointer to a newly initialized structure is passed in the function call, the function searches from the beginning of the GC_PARM_BLK; if the structure contains data from a previously found parameter, the function searches from that parameter onward. When the function completes successfully, the structure is updated to contain retrieved information for the parameter that was found.

Cautions

Unlike the similar **gc_util_find_parm( )** function, the **parm** pointer used in this function *cannot* be used to update the parameter itself because it points to a data structure that is in the application's memory rather than a location in the GC_PARM_BLK itself.

Errors

If this function returns GC_ERROR 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"
#include "gcip.h"

void search_parm_block(GC_PARM_BLKP parm_blkp)
{
   GC_PARM_DATA_EXT parm_data_ext;
   int ret = 0;

   /* Initialize this structure for two reasons:
    * 1. To search from the first parameter in the parm block
    * 2. The first time this structure is used it must be initialized
    */
   INIT_GC_PARM_DATA_EXT(&parm_data_ext);

   /* loop to retrieve all of the parameters and associated data in the
    * GC_PARM_BLK that match the set_ID/parm_ID pair for SIP header fields.
    */
   while ( GC_SUCCESS == (ret = gc_util_find_parm_ex(parm_blkp, IPSET_SIP_MSGINFO,
                                                     IPPARM_SIP_HDR, &parm_data_ext)) )
   {
      /* process GC_PARM_DATA_EXT structure */
      .
      .
      .
   }
```

```
    /* Check for error */
    if ( GC_ERROR == ret)
    {
        /* process error */
    }

    .
    .
    .
}
```

■ See Also

- **gc_util_find_parm( )**
- **gc_util_next_parm_ex( )**

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

■ 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
parm_blkpp	points to the address of a valid GC_PARM_BLK where the parameter is to be inserted. Set ***parm_blkpp** to NULL if the parameter is to be inserted into a new block.
setID	set ID of the parameter to be inserted
parmID	parm ID of the parameter to be inserted
data_size	size, in bytes, of the data associated with this parameter
datap	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"
#include "gcip.h"
#include "gcip_defs.h"

void main( )
{
    GC_PARM_BLKP 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_delete_parm_blk( )**
- **gc_util_insert_parm_val( )**

gc_util_insert_parm_ref_ex( )

Name: int gc_util_insert_parm_ref_ex(parm_blkpp, setID, parmID, data_size, datap)

Inputs:

GC_PARM_BLKP *parm_blkpp	• pointer to the address of a valid GC_PARM_BLK
unsigned long setID	• set ID of parameter to be inserted
unsigned long parmID	• parm ID of parameter to be inserted
unsigned long data_size	• size in bytes of the parameter data
void *datap	• pointer to the parameter data

Returns: GC_SUCCESS if successful
GC_ERROR if failure

Includes: gclib.h
gcerr.h

Category: GC_PARM_BLK utility

Mode: synchronous

Description

The **gc_util_insert_parm_ref_ex**( ) function inserts a parameter into a GC_PARM_BLK data structure by reference.

The **gc_util_insert_parm_ref_ex**( ) function **must** be used rather than the similar **gc_util_insert_parm_ref**( ) function whenever the parameter data exceeds 255 bytes in length. The **gc_util_insert_parm_ref_ex**( ) function is backwards compatible and can be used with any setID/parmID pair regardless of whether that pair supports data lengths greater than 255 bytes.

Note: The only Global Call parameters that currently support data longer than 255 bytes are IPSET_SIP_MSGINFO/IPPARM_SIP_HDR, which is used for SIP message headers, and the parameters in the IPSET_SDP set, which is used in 3PCC mode.

A new GC_PARM_BLK can be created by inserting the first parameter with ***parm_blkpp** set to NULL. A parameter can be inserted in an existing GC_PARM_BLK by setting ***parm_blkpp** to the address of that block.

Note: Parameters are contained in the GC_PARM_BLK in the order in which they are inserted, and they will also be retrieved via the **gc_util_next_parm_ex( )** function in the same order.

Parameter	Description
parm_blkpp	points to the address of a valid GC_PARM_BLK where the parameter is to be inserted. Set ***parm_blkpp** to NULL to insert the parameter into a new block.
setID	parameter set ID of the parameter to be inserted
parmID	parameter ID of the parameter to be inserted

Parameter	Description
data_size	size, in bytes, of the data associated with this parameter
datap	points to the data associated with this parameter

Cautions

- To avoid a memory leak, any GC_PARM_BLK created must be deleted using the **gc_util_delete_parm_blk( )** function.
- Insertion of data that exceeds 255 bytes in length is only supported for specific setID/parmID pairs. Refer to the appropriate Global Call Technology Guide for information on maximum data length for each setID/parmID pair.

Errors

- If this function returns GC_ERROR 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.
- Attempting to insert data greater than 255 bytes in length using a setID/parmID pair that does not support large data sizes produces an error indication. In this situation, the **gc_ErrorInfo( )** function returns the value EGC_INVPARM.

Example

```
#include "gclib.h"
#include "gcip.h"

void SetHeader(void)
{
   GC_PARM_BLKP my_blkp = NULL;
   char* pChar = "Remote-Party_ID: This string can be greater than 255 bytes";

   /* Add 1 to strlen for null termination */
   unsigned long data_size = strlen(pChar) + 1;

   /* insert parm and associated data into the GC_PARM_BLK */
   if ( gc_util_insert_parm_ref_ex( &my_blkp, IPSET_SIP_MSGINFO, IPPARM_SIP_HDR, data_size,
                                    (void*)( pChar )) != GC_SUCCESS )
   {
      /* Process error */
   }

   /* At this point the application can overwrite the data pointed to by pChar. */
   pChar = NULL;

   /* Pass the parm block to GC */
   if ( gc_SetUserInfo( GCTGT_GCLIB_CRN, crn, &my_blkp, GC_SINGLECALL) != GC_SUCCESS )
   {
      /* Process error */
   }

   /* GC_PARM_BLK is no longer needed; delete the block */
   gc_util_delete_parm_blk( my_blkp );
}
```

See Also

- **gc_util_delete_parm_blk( )**
- **gc_util_insert_parm_ref( )**
- **gc_util_insert_parm_val( )**

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

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
parm_blkpp	points to the address of a valid GC_PARM_BLK where the parameter is to be inserted. Set ***parm_blkpp** to NULL if the parameter is to be inserted into a new block.
setID	set ID of the parameter to be inserted
parmID	parm ID of the parameter to be inserted
data_size	size, in bytes, of the data associated with this parameter
data	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"
#include "gcip.h"
#include "gcip_defs.h"

void main( )
{
    GC_PARM_BLKP 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_delete_parm_blk( )**
- **gc_util_insert_parm_ref( )**

gc_util_next_parm( )

Name: GC_PARM_DATAP gc_util_next_parm(parm_blk, cur_parm)

Inputs: GC_PARM_BLKP *parm_blk • pointer to GC_PARM_BLK

GC_PARM_DATAP *cur_parm • pointer to current parameter data structure

Returns: a pointer if successful
NULL if no more parameters exist

Includes: gclib.h
gcerr.h

Category: GC_PARM_BLK utility

Mode: synchronous

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

None

Example

```
#include "gclib.h"
#include "gcip.h"
#include "gcip_defs.h"
```

```
void main( )
{
    GC_PARM_BLKP 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_util_next_parm_ex( )

Name: int gc_util_next_parm_ex(parm_blk, parm)

Inputs: GC_PARM_BLKP parm_blk • pointer to GC_PARM_BLK

GC_PARM_DATA_EXTP parm • pointer to structure identifying current parameter

Outputs: GC_PARM_DATA_EXTP parm • pointer to structure identifying next parameter

Returns: GC_SUCCESS if successful
EGC_NO_MORE_PARMS if no more parameters exist in the GC_PARM_BLK
GC_ERROR if failure

Includes: gclib.h
gcerr.h

Category: GC_PARM_BLK utility

Mode: synchronous

■ Description

The **gc_util_next_parm_ext( )** function is used to retrieve the next parameter (relative to a specified current parameter) from a GC_PARM_BLK in the form of a GC_PARM_DATA_EX data structure. Calling this function repetitively and passing a pointer to the GC_PARM_DATA_EX structure that was returned by the previous call allows an application to sequentially retrieve all of the parameters in a GC_PARM_BLK.

This function **must** be used instead of **gc_util_next_parm( )** if the parameter data can potentially exceed 255 bytes. This function is backward compatible and can be used instead of **gc_util_next_parm( )** for any GC_PARM_BLK, regardless of whether the parameter block contains setID/parmID pairs that support data lengths greater than 255 bytes.

Note: The only Global Call parameters that currently support data longer than 255 bytes are IPSET_SIP_MSGINFO/IPPARM_SIP_HDR, which is used for SIP message headers, and the parameters in the IPSET_SDP set, which is used in 3PCC mode.

The **gc_util_next_parm_ex( )** function updates the data structure pointed to by the **parm** pointer and returns GC_SUCCESS if there is another parameter in the GC_PARM_BLK following the current parameter that was identified in the function call. If the current parameter identified by **parm** is the last parameter in the GC_PARM_BLK, the function returns EGC_NO_MORE_PARMS.

Parameter	Description
parm_blk	points to the valid GC_PARM_BLK structure where data is stored
parm	pointer to a valid GC_PARM_DATA_EXT structure provided by the application. If the pointer that is passed in the function call points to a structure that was just initialized with INIT_GC_PARM_DATA_EXT(parm), the function retrieves the first parameter in the GC_PARM_BLK. If the passed pointer points to a structure that contains data from a previously found parameter, the function retrieves the next parameter in the block (if any). When the function completes successfully, the GC_PARM_DATA_EXT structure is updated to contain retrieved information for the parameter.

Cautions

Unlike the similar **gc_util_next_parm( )** function, the **parm** pointer used in this function *cannot* be used to update the parameter itself because it points to a data structure that is in the application's memory rather than a location in the GC_PARM_BLK itself.

Errors

If this function returns GC_ERROR 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"
#include "gcip.h"

void process_parm_block(GC_PARM_BLKP pparm_blk)
{
   GC_PARM_DATA_EXT parm_data_ext;
   int ret = 0;

   /* Initialize this structure for two reasons:
    * 1. To retrieve the first parameter in the parm block
    * 2. The first time this structure is used it must be initialized
    */
   INIT_GC_PARM_DATA_EXT(&parm_data_ext);

   /* Loop to retrieve all of the parameters and associated data from
    * the GC_PARM_BLK
    */insert parm by reference */
   while ( GC_SUCCESS == (ret = gc_util_next_parm_ex( pparm_blk, &parm_dat_ext)) )
   {
      /* Process set_ID/parm_ID pairs */
      switch(parm_data_ext.set_ID);
      {
         .
         .
         .
      }
   }
```

```
    /* Check for error */
    if ( GC_ERROR == ret )
    {
        /* Process error */
    }

    .
    .
    .
}
```

■ See Also

- **gc_util_find_parm_ex( )**
- **gc_util_next_parm( )**

gc_WaitCall( )

Name: int gc_WaitCall(linedev, crnp, waitcallp, timeout, mode)

Inputs:

LINEDEV linedev	• Global Call line device handle
CRN *crnp	• pointer to CRN
GC_WAITCALL_BLK *waitcallp	• reserved for future use
int timeout	• time-out
unsigned long mode	• async

Returns: 0 if successful
<0 if failure

Includes: gclib.h
gcerr.h

Category: basic

Mode: asynchronous

■ 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.

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.

Parameter	Description
linedev	Global Call line device handle
crnp	must be set to NULL in asynchronous mode. When the GCEV_OFFERED event is received, the CRN can be retrieved.
waitcallp	not used. Set to zero.
timeout	ignored in asynchronous mode
mode	set to EV_ASYNC for asynchronous execution

Termination Events

None

Note: 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.
- The CRN will not be available until an inbound call has arrived (that is, until the GCEV_OFFERED event is received).
- 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 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>
#include <gcip.h>
#include <gcip_defs.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 */
        /* process error return as shown */
        gc_ErrorInfo( &gc_error_info );
        printf ("Error: gc_WaitCall() 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);
        }

    }
    /*
     * GCEV_OFFERED event will indicate incoming call has arrived.
     */
    return (0);
}
```

See Also

- **gc_DropCall( )**
- **gc_MakeCall( )**
- **gc_ReleaseCallEx( )**
- **gc_ResetLineDev( )**

Events 3

This chapter provides information about the events used by the Global Call software with Host Media Processing (HMP). Topics include:

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 231 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.

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_ACCEPT_INIT_XFER

Termination event for **gc_AcceptInitXfer( )**

Associated with a call (CRN)

Indicates that the function was successful, that is, party C successfully accepted the initiate transfer request from remote party A.

GCEV_ACCEPT_INIT_XFER_FAIL

Termination event for **gc_AcceptInitXfer( )**

Associated with a call (CRN)

The **gc_AcceptInitXfer( )** function failed.

GCEV_ACCEPT_XFER

Termination event for **gc_AcceptXfer( )**

Associated with a call (CRN)

Indicates that the function was successful, that is, the local party successfully accepted the call transfer request from remote party A.

GCEV_ACCEPT_XFER_FAIL

Termination event for **gc_AcceptXfer( )**

Associated with a call (CRN)

The **gc_AcceptXfer( )** function failed.

GCEV_ACKCALL

Termination event for **gc_CallAck( )**

Associated with a call (CRN)

Indicates termination of **gc_CallAck( )**.

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_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_AttachResource( )**

Associated with a line device (LDID)

The corresponding attach function failed.

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.

GCEV_CONNECTED (see also below)

Termination event for **gc_MakeCall( )**

Associated with a call (CRN)

Call is connected.

GCEV_CONNECTED (see also above)

Unsolicited event

Associated with a call (CRN)

Generated during a call transfer when a call is put "on hold pending transfer" and the new call that is being set up is terminated.

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_DISCONNECTED (see also below)

Unsolicited event

Associated with a call (CRN)

Call is disconnected by remote end.

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_DROPCALL

Termination event for **gc_DropCall( )**

Associated with a call (CRN)

Call is disconnected and enters Idle state.

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_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_INIT_XFER

Termination event for **gc_InitXfer( )**

Associated with a call (CRN)

Indicates that the function was successful (implies that the request was accepted by remote party C).

GCEV_INIT_XFER_FAIL

Termination event for **gc_InitXfer( )**

Associated with a call (CRN)

The **gc_InitXfer( )** function failed.

GCEV_INIT_XFER_REJ

Termination event for **gc_InitXfer( )**

Associated with a call (CRN)

Indicates that the initiate transfer request was successfully invoked by party A via the **gc_InitXfer( )** function, but the request was rejected by remote party C.

GCEV_INVOKE_XFER

Termination event for **gc_InvokeXfer( )**

Associated with a call (CRN)

The call transfer was successfully completed by the **gc_InvokeXfer( )** function.

GCEV_INVOKE_XFER_FAIL

Termination event for **gc_InvokeXfer( )**

Associated with a call (CRN)

The **gc_InvokeXfer( )** function failed.

GCEV_INVOKE_XFER_REJ

Termination event for **gc_InvokeXfer( )**

Associated with a call (CRN)

Indicates that the call transfer request was successfully invoked by party A via the **gc_InvokeXfer( )** function, but the request was rejected by the remote party that received the call transfer request.

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_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_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_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_REJ_INIT_XFER

Termination event for **gc_RejectInitXfer( )**

Associated with a call (CRN)

Indicates that the function was successful, that is, party C successfully rejected the initiate transfer request from remote party A.

GCEV_REJ_INIT_XFER_FAIL

Termination event for **gc_RejectInitXfer( )**

Associated with a call (CRN)

The **gc_RejectInitXfer( )** function failed.

GCEV_REJ_XFER

Termination event for **gc_RejectXfer( )**

Associated with a call (CRN)

Indicates that the function was successful, that is, the local party successfully rejected the call transfer request from the remote party.

GCEV_REJ_XFER_FAIL

Termination event for **gc_RejectXfer( )**

Associated with a call (CRN)

The **gc_RejectXfer( )** function failed.

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_REQ_INIT_XFER

Unsolicited event

Associated with a call (CRN)

Notifies application at party C of the initiate transfer request from remote party A.

GCEV_REQ_XFER

Unsolicited event

Associated with a call (CRN)

Notifies application at the local party receiving the call transfer request from remote party A.

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_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_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_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_TASKFAIL

Termination event for **gc_AcceptCall( )**, **gc_AnswerCall( )**, **gc_CallAck( )**, **gc_DropCall( )**, **gc_Extension( )**,**gc_Listen( )**, **gc_MakeCall( )**, **gc_ReqService( )**, **gc_RespService( )**, **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_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_XFER_CMPLT

Unsolicited event

Associated with a call (CRN)

Notifies application at the local party accepting the call transfer request that the call transfer was completed (that is, the call between party B and C was connected).

GCEV_XFER_FAIL

Unsolicited event

Associated with a call (CRN)

Notifies application at the local party accepting the call transfer request that the call transfer was not completed because of a failure (time-out, busy, no answer, etc.).

Data Structures 4

This chapter provides an alphabetical reference to the data structures used by the Global Call software with Host Media Processing (HMP). The following data structures are described:

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_PARMS];
} 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 245 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                          n_asos;
    ALARM_SOURCE_OBJECT_FIELD    aso_fields[MAX_ASOS];
} ALARM_SOURCE_OBJECT_LIST;
```

■ 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_CUSTOM1_LIB" – Custom call control library 1
- "GC_CUSTOM2_LIB" – Custom call control library 2
- "GC_DM3CC_LIB" – DM3CC call control library
- "GC_H3R_LIB" – IP 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;       /* product ID */
   unsigned char  ct_devfamily;    /* device family */
   unsigned char  ct_devmode;      /* device mode */
   unsigned char  ct_nettype;      /* network interface */
   unsigned char  ct_busmode;      /* bus architecture */
   unsigned char  ct_busencoding;  /* bus encoding */
   union {
           unsigned char ct_RFU[7];     /* reserved */
           struct {
                 unsigned char ct_prottype;
           } ct_net_devinfo;
   } ct_ext_devinfo;
} CT_DEVINFO;
```

■ Description

The CT_DEVINFO structure contains information about a specified line device.

Valid values for each member of the structure are defined in *ctinfo.h*, which is referenced by *gclib.h*.

■ Field Descriptions

See the *Voice API for Host Media Processing Library Reference* for further information about the CT_DEVINFO data structure when used with HMP.

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 *Global Call IP for Host Media Processing Technology 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.

■ 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

  See Table 9 for descriptions of the fields in each type of structure.

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 9. Service Type Data Structure Field Descriptions

Service Type	Data Structure Field in Union	Description
GCACK_SERVICE_INFO	info	structure containing type and length of information requested
	info.info_type	type of information requested. The possible values are: • DESTINATION_ADDRESS – request more DNIS • ORIGINATION_ADDRESS – request more ANI
	info.info_len	length of the information requested (typically number of digits)

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_CUSTOM1_LIB" – Custom call control library 1
- "GC_CUSTOM2_LIB" – Custom call control library 2
- "GC_DM3CC_LIB" – DM3CC call control library
- "GC_H3R_LIB" – IP 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

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.

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_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_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_BLKP;
```

■ 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 13, "Possible Set ID, Parm ID Pairs used in GCLIB_MAKECALL_BLK Structure", on page 290 or the *Global Call IP for Host Media Processing Technology 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 288 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_xxx). 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_DATAP;
```

Description

The GC_PARM_DATA structure contains parameter data. Consult the *Global Call IP for Host Media Processing Technology 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 288 for an illustration of the memory block format. See Table 13, "Possible Set ID, Parm ID Pairs used in GCLIB_MAKECALL_BLK Structure", on page 290 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 *Global Call IP for Host Media Processing Technology 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_DATA_EXT

```
typedef struct
{
    unsigned long     version;
    void*             pInternal;
    unsigned long     set_ID;
    unsigned long     parm_ID;
    unsigned long     data_size;
    void*             pData;
}GC_PARM_DATA_EXT, *GC_PARM_DATA_EXTP;
```

Description

The GC_PARM_DATA_EXT structure contains parameter data retrieved from a GC_PARM_BLK by the **gc_util_find_parm_ex( )** and **gc_util_next_parm_ex( )** functions. These functions were added to the Global Call API library to support the retrieval of parameters that may exceed 255 bytes in length, but these functions always return the retrieved parameter information in a GC_PARM_DATA_EX structure regardless of whether the parameter data exceeds 255 bytes.

The set ID and parm ID as a pair identify the parameter. Set IDs and parm IDs that are common to multiple Global Call technologies are listed in the *Global Call API Library Reference*, and additional technology-specific parameters are listed in each of the various Global Call Technology Guides. Unless a particular set ID/parm IP pair specifically indicates that it supports parameter data that exceeds 255 bytes in length, users should assume that the parameter data length does not exceed 255.

The list of Global Call IP parameters that support long data currently includes:

- IPSET_SIP_MSGINFO / IPPARM_SIP_HDR (SIP message headers)

Note: The only Global Call parameters that currently support data longer than 255 bytes are IPSET_SIP_MSGINFO / IPPARM_SIP_HDR, which is used for SIP message headers, and the parameters in the IPSET_SDP set, which is used in 3PCC mode.

Applications should use the **INIT_GC_PARM_DATA_EXT**( ) function to initialize the structure with the correct version number and default field values before setting the appropriate values.

Field Descriptions

The fields of GC_PARM_DATA_EXT are described as follows:

version
: identifies the version of the data structure implementation. This field is reserved for library use and should **not** be modified by applications.

pInternal
: pointer used to identify position within the GC_PARM_BLK structure. This field is reserved for library use and should **not** be used or modified by applications.

set_id
: the set ID of the retrieved parameter

parm_id
: the parameter ID of the retrieved parameter

data_size
: the size of the retrieved parameter data in bytes

pData
: pointer to the first byte of the parameter value buffer.

GC_REROUTING_INFO

```
typedef struct
{
      char*              rerouting_num;    /*Rerouting number, terminated with '\0' */
      GCLIB_ADDRESS_BLK  rerouting_addr;   /*Rerouting address */
      GC_PARM_BLK        *parm_blkp;       /*Additional parameters associated*/
} GC_REROUTING_INFO, *GC_REROUTING_INFOP;
```

Description

The GC_REROUTING_INFO structure contains the rerouting information for call transfer. It is used as event data for GCEV_REQ_XFER. The structure is sent to the application along with the GCEV_REQ_XFER event and is accessed by dereferencing the extevtdatap pointer within the METAEVENT structure.

Note: This structure contains the information only until the next call of **gc_GetMetaEvent( )** or **gc_GetMetaEventEx( )**. All GC_REROUTING_INFO structure data must be processed or cached within the application, or risk being lost upon the next call of **gc_GetMetaEvent( )** or **gc_GetMetaEventEx( )**.

Field Descriptions

The fields of GC_REROUTING_INFO are described as follows:

rerouting_num
: a null terminated string of maximum size GC_ADDRSIZE (128) bytes

rerouting_addr
: rerouting address information. See the GCLIB_ADDRESS_BLK structure.

Note: The three subaddress elements of the GCLIB_ADDRESS_BLK structure are unused for IP applications.

*parm_blkp
: a pointer to the parm_blk structure

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_BLKP    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.

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 *Global Call IP for Host Media Processing Technology 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. See the *Global Call IP for Host Media Processing Technology Guide* for more information.

sub_address
: specifies the destination sub-address, typically digits. The format is technology-specific. See the *Global Call IP for Host Media Processing Technology 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

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_BLKP         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 GCMKCALLBLK_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 13, "Possible Set ID, Parm ID Pairs used in GCLIB_MAKECALL_BLK Structure", on page 290. See the *Global Call IP for Host Media Processing Technology 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             rfu1;
} 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.

Error Codes 5

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_AGGETXMITSLOT
: **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_DTGETXMITSLOT
dt_getxmitslot( ) function failed

EGC_DTLISTEN
Network interface (dti) device listen failed

EGC_DTOPEN
dt_open( ) function failed

EGC_DTUNLISTEN
Network interface (dti) device unlisten failed

EGC_DUPENTRY
Duplicate entry inserted into Global Call database

EGC_DXGETXMITSLOT
dx_getxmitslot( ) function failed

EGC_DXLISTEN
Voice device listen failed

EGC_DXOPEN
dx_open( ) 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_GETXMITSLOT
gc_GetXmitSlot( ) function failed

EGC_GLARE
gc_MakeCall( ) 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_INVDATABUFFSIZE
(RTCM) Invalid parm data buffer size

EGC_INVDEVNAME
Invalid device name

EGC_INVLINEDEV
Invalid line device passed

EGC_INVMETAEVENT
Invalid metaevent

EGC_INVPARM
Invalid parameter (argument)

EGC_INVPARM_CCLIB
(RTCM) Invalid parm to CCLib

EGC_INVPARM_FIRMWARE
(RTCM) Invalid parm to firmware

EGC_INVPARM_GCLIB
(RTCM) Invalid parm to GCLib

EGC_INVPARM_PROTOCOL
(RTCM) Invalid parm to protocol

EGC_INVPARM_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
gc_Listen( ) 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 **gc_AttachResource( )** 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
gc_UnListen( ) 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 call control library-specific reason

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 **gc_AttachResource( )** 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_RESETABLE_ FATALERROR
Resetable 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:

6.1 Alarm Source Object IDs

This section contains information pertaining to the Global Call Alarm Management System (GCAMS).

Table 10 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 *Global Call IP for Host Media Processing Technology Guide*.

Table 10. Alarm Source Object IDs

Alarm Source Object ID	Description
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 11 shows the combinations of physical or logical entities and software module entities that can make up a target (**target_type**).

Table 11. Supported Target Types

Software Module	Entity			
	System	Network Interface	Channel	CRN
GCLib	S	S	S	S
CCLib		S	S	S
S = Supported				

Table 12 shows the target types, as described in Table 11, with various target IDs to represent valid target objects.

Table 12. Target Type and Target ID Pairs

Target Type	Target ID	Description
GCTGT_GCLIB_SYSTEM	GCGV_LIB(0)	Global Call library module target object
GCTGT_GCLIB_NETIF	Global Call Line device ID	Network Interface target object in Global Call library module
GCTGT_CCLIB_NETIF	Global Call Line device ID	Network Interface target object in call control library module
GCTGT_GCLIB_CHAN	Global Call Line device ID	Channel target object in Global Call library module
GCTGT_CCLIB_CHAN	Global Call Line device ID	Channel target object in call control library module
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

Figure 2 gives the memory diagram for the GC_PARM_BLK structure.

Figure 2. GC_PARM_BLK Memory Diagram

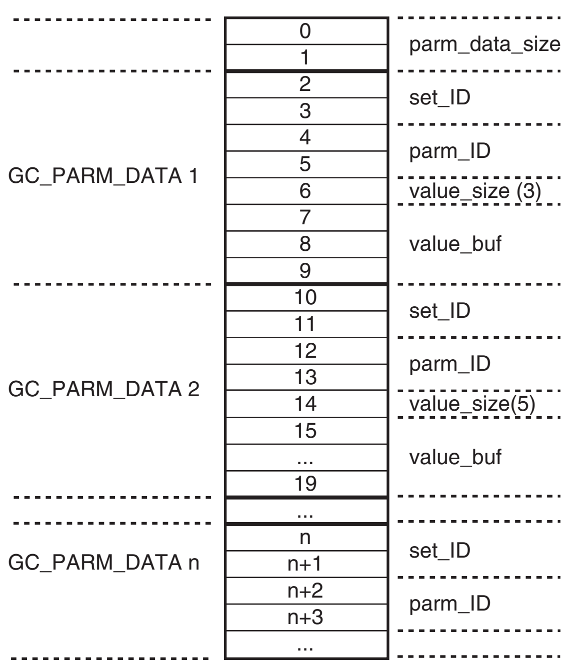


Figure 3 is an example of the GC_PARM_BLK structure filled with some parameters for IP technology.

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 13 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. See the *Global Call IP for Host Media Processing Technology Guide* for technology-specific information.

Table 13. 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 13. 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( )** function are shown in Table 14 and Table 15.

Table 14. Global Call 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 14. Global Call 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	CClib ID	long	GC_R_O
GCSET_ CCLIB_INFO	GCPARM_ CCLIB_NAME	GCTGT_ GCLIB_NETIF GCTGT_ GCLIB_CHAN	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 15. 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 15 are examples only. The configurable parameters depend on the CCLib implementation.

Index

F

G

L

M

N

O

R

S

T

U

V