Porting Global Call H.323 Applications from Embedded Stack to Host-Based Stack

Application Note

05-1857-002

Publication Date: November, 2002

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

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

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

For **Sales Offices**, visit the Intel Telecom Building Blocks Sales Offices page:
http://www.intel.com/network/csp/sales/

Table of Contents

1. Introduction

This technical note provides information for application developers on porting existing Global Call applications that use the embedded H.323 IP stack provided in System Release 5.x to Global Call applications that use the host-based H.323 stack provided in System Release 6.0 and later. Topics include:

- Deprecated Global Call functions and the preferred equivalents
- Run time configuration
- The relevant Global Call API functions that replace the *config.val* file
- Compatibility issues to be fixed in future system releases
- Compatibility issues
- Enhanced functionality available in the System Release 6.0

1.1. Related Documentation

See the following documentation for related information:

- *Global Call API Programming Guide*
- *Global Call API Library Reference*
- *Global Call IP Technology User's Guide*

2. Deprecated Global Call Functions

The following Global Call functions, which may have been used in applications developed using the NetTSC embedded stack implementation are deprecated in System Release 6.0. The preferred equivalent for each deprecated function is given in *Table 1*.

Table 1. Deprecated Global Call Functions

Function	**Preferred Equivalent**
gc_Attach( )	**gc_AttachResource( )**
gc_ErrorValue( )	**gc_ErrorInfo( )**
gc_GetANI( )	**gc_GetCallInfo( )**
gc_GetDNIS( )	**gc_GetCallInfo( )**
gc_GetNetworkH( )	**gc_GetResourceH( )**
gc_GetVoiceH( )	**gc_GetResourceH( )**
gc_Open( )	**gc_OpenEx( )**
gc_ReleaseCall( )	**gc_ReleaseCallEx( )**
gc_ResultMsg( )	**gc_ResultInfo( )/gc_ErrorInfo( )**
gc_SetCallingNum( )	**gc_SetConfigData( )**
gc_SetEvtMsk( )	**gc_SetConfigData( )**
gc_SndMsg( )	**gc_SetUserInfo( )**

See the *Global Call API Library Reference* for generic information about these functions and the *Global Call IP Technology User's Guide* for technology-specific information.

2.1. Event Masking

The **gc_SetEvtMsk( )** function is deprecated in System Release 6.0. The preferred equivalent is the **gc_SetConfigData( )** function. The use of **gc_SetConfigData( )** is different depending on the type of events being masked as follows:

- Call state events, for example GCEV_ALERTING, are masked using **gc_SetConfigData( )** with a target type of GCTGT_GCLIB_CHAN and a target ID equal to a **line device**. Global Call uses this generic mechanism to mask call state events in all technologies. See the section on *Call State Event Configuration* in the *Global Call API Programming Guide* for more information on masking call state events.

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

- GCEV_EXTENSION events, for example notification of received DTMF digits, are masked using the **gc_SetConfigData( )** function with a target type of GCTGT_CCLIB_NETIF and a target ID equal to a **board device**. For IP technology, Global Call uses this mechanism to enable or disable events for all line devices associated with a specific board device. See the section on *Enabling and Disabling Unsolicited Notification Events* in the *Global Call IP Technology User's Guide* for more information on masking GCEV_EXTENSION events.

3. Run Time Configuration

The following features, which in System Release 5.x were configurable at download time, are now configurable at run time using the Global Call API.

- IP Address and Port Specification
- DTMF Transfer Mode Configuration

3.1. IP Address and Port Specification

When using Global Call and the host based stack implementation, IP address and port information is configurable in the IPCCLIB_START_DATA structure used by the **gc_Start( )** function. Each NIC or NIC address (if a NIC supports multiple addresses) corresponds to a IPT board device. For each IPT board device, a IP_VIRTBOARD structure contains the address and port variables. The local IP address for each IPT board device is a parameter of type IPADDR in the IP_VIRTBOARD structure. The signaling ports used for H.323 and SIP are also parameters in the IP_VIRTBOARD structure.

Caution:

In H.323, by default, the RAS signaling port is assigned to be one less than the H.323 signaling port. Consequently, to avoid a port conflict when configuring multiple boards, do not assign consecutive H.323 signaling port numbers to boards.

The following code example shows how to specify address and port information for a NIC with one address. The bold text shows the most relevant lines of code:

```
#define BOARDS_NUM   1

/* initialize start parameters */
IPCCLIB_START_DATA cclibStartData;
memset(&cclibStartData,0,sizeof(IPCCLIB_START_DATA));

IP_VIRTBOARD virtBoards[BOARDS_NUM];
memset(virtBoards,0,sizeof(IP_VIRTBOARD)*BOARDS_NUM);
```

```
cclibStartData.version     = 0x0100;   // must be set to 0x0100
cclibStartData.delimiter   = ',';
cclibStartData.num_boards  = BOARDS_NUM;
cclibStartData.board_list  = virtBoards;

virtBoards[0].total_max_calls = IP_CFG_MAX_AVAILABLE_CALLS;
virtBoards[0].h323_max_calls = IP_CFG_MAX_AVAILABLE_CALLS;
virtBoards[0].sip_max_calls = IP_CFG_MAX_AVAILABLE_CALLS;
virtBoards[0].localIP.ip_ver = IPVER4;    // must be set to IPVER4
virtBoards[0].localIP.u_ipaddr.ipv4 = IP_CFG_DEFAULT;
virtBoards[0].h323_signaling_port = IP_CFG_DEFAULT;
virtBoards[0].sip_signaling_port = IP_CFG_DEFAULT;
virtBoards[0].reserved = NULL;            // must be set to NULL

CCLIB_START_STRUCT cclibStartStruct[] = {
    {"GC_IPM_LIB", NULL},
    {"GC_H3R_LIB", &cclibStartData}
};

GC_START_STRUCT gcStartStruct;
gcStartStruct.cclib_list = cclibStartStruct;
gcStartStruct.num_cclibs = 2;
int rc = gc_Start(&gcStartStruct);
if(GC_SUCCESS != rc)
{
    // handle the error
}
```

NOTE: The IP_CFG_DEFAULT define indicates to the call control library that it should determine and fill in the correct values. IP_CFG_DEFAULT values can be replaced by discrete values.

If two Global Call IP applications are required to be running on the same machine (using same stack), the second application must modify the IPCCLIB_START_DATA structure to have a different h323_signaling_port value so that there is no conflict. If there is a conflict, the IP call control library will fail to load.

3.2. DTMF Transfer Mode Configuration

In System Release 5.x, the DTMF transfer mode was configurable by setting the value of the **PrmDTMFXferMode** parameter in the *.config* file. In System Release 6.0, the DTMF transfer mode is configurable at run time using the Global Call API. The DTMF mode can be specified for all line devices (using the **gc_SetConfigData( )** function) or on a per line device basis (using **gc_SetUserInfo( )** with the duration parameter set to GC_ALLCALLS).

The GC_PARM_BLK associated with the **gc_SetConfigData( )** or **gc_SetUserInfo( )** function is used to specify the DTMF modes that are supported. The GC_PARM_BLK must include the following parameter set ID and parameter ID:

- IPSET_DTMF
    - IPPARM_SUPPORT_DTMF_BITMASK which can be one of the following values:
        - IP_DTMF_TYPE_ALPHANUMERIC_DTMF (H.323 only)
        - IP_DTMF_TYPE_INBAND_RTP
        - IP_DTMF_TYPE_RFC_2833

See the *Global Call IP Technology User's Guide* for more detail.

4. config.val File Redundancy

In System Release 5.x, the *config.val* file contained the following configurable information:

- Stack allocation parameters
- Address string delimiter
- Vendor information
- Registration, Admission and Status (RAS) information
- Coder information

In System Release 6.0, the *config.val* file is **not** used. The following sections describe how Global Call API functions provide the configuration functionality previously available in the *config.val* file.

4.1. Setting Stack Allocation Parameters

Stack allocation parameters are configurable in the IPCCLIB_START_DATA structure used by the **gc_Start( )** function. The parameters are configurable on a virtual board basis. Since the underlying call control library now supports both the H.323 and SIP protocols, parameters are included for each protocol. The parameters include:

- **total_max_calls** - The maximum total number of IPT devices that can be open concurrently.
- **h323_max_calls** - The maximum number of IPT devices used for H.323 calls
- **sip_max_calls** - The maximum number of IPT devices used for SIP calls.

Refer to the IP_VIRTBOARD data structure description in the *Global Call IP Technology User's Guide* for a complete description of the structure. The following code example shows how to specify stack allocation parameters for a NIC with one address. The bold text shows the most relevant lines of code:

```
#define BOARDS_NUM   1
/* initialize start parameters */
IPCCLIB_START_DATA cclibStartData;
memset(&cclibStartData,0,sizeof(IPCCLIB_START_DATA));

IP_VIRTBOARD virtBoards[BOARDS_NUM];
memset(virtBoards,0,sizeof(IP_VIRTBOARD)*BOARDS_NUM);

cclibStartData.version     = 0x0100;   // must be set to 0x0100
cclibStartData.delimiter   = ',';
cclibStartData.num_boards  = BOARDS_NUM;
cclibStartData.board_list  = virtBoards;

virtBoards[0].total_max_calls = IP_CFG_MAX_AVAILABLE_CALLS;
virtBoards[0].h323_max_calls = IP_CFG_MAX_AVAILABLE_CALLS;
virtBoards[0].sip_max_calls = IP_CFG_MAX_AVAILABLE_CALLS;
virtBoards[0].localIP.ip_ver = IPVER4;    // must be set to IPVER4
virtBoards[0].localIP.u_ipaddr.ipv4 = IP_CFG_DEFAULT;
virtBoards[0].h323_signaling_port = IP_CFG_DEFAULT;
virtBoards[0].sip_signaling_port = IP_CFG_DEFAULT;
virtBoards[0].reserved = NULL;           // must be set to NULL

CCLIB_START_STRUCT cclibStartStruct[] = {
    {"GC_IPM_LIB", NULL},
    {"GC_H3R_LIB", &cclibStartData}
};

GC_START_STRUCT gcStartStruct;
gcStartStruct.cclib_list = cclibStartStruct;
gcStartStruct.num_cclibs = 2;
int rc = gc_Start(&gcStartStruct);
if(GC_SUCCESS != rc)
{
    // handle the error
}
```

NOTE: The IP_CFG_MAX_AVAILABLE_CALLS define indicates to the call control library that it should determine and fill in the correct values. Discrete values can also be used. Values must be in the range 0 to 2016. Specifying values less than the maximum reduces the amount of memory allocated by the stack and helps improve performance.

4.2. Setting the Address String Delimiter

The delimiter that separates different parts of the address in composite address strings can be set in the IPCCLIB_START_DATA structure used by the **gc_Start( )** function. The parameter is configurable on a system-wide basis. The following code segment shows how to set the delimiter to be a comma (,).

```
#define BOARDS_NUM   1

/* initialize start parameters */
IPCCLIB_START_DATA cclibStartData;
memset(&cclibStartData,0,sizeof(IPCCLIB_START_DATA));
```

```
cclibStartData.version     = 0x0100;   // must be set to 0x0100
cclibStartData.delimiter   = ',';
cclibStartData.num_boards  = BOARDS_NUM;
cclibStartData.board_list  = virtBoards;
```

4.3. Setting Vendor Information

The application can set vendor information using the **gc_SetConfigData( )** function. The following are the relevant function parameter values:

- **target_type** - GCTGT_CCLIB_NETIF
- **target_id** - IPT board device
- **target_datap** - A pointer to a GC_PARM_BLK structure that contains configuration information including the vendor information.

To specify vendor information, the GC_PARM_BLK must contain the following parameter set ID and parameter IDs:

- IPSET_VENDORINFO
  - IPPARM_VENDOR_PRODUCT_ID - Product ID. A string with a maximum length of MAX_PRODUCT_ID_LENGTH (32) characters.
  - IPPARM_VENDOR_VERSION_ID - Version ID. A string with a maximum length of MAX_VERSION_ID_LENGTH (32) characters.
  - IPPARM_H221NONSTD - Non-standard vendor information. A structure containing country code, extension, and manufacturer's code. See the IP_H221NONSTANDARD structure description in the *Global Call IP Technology User's Guide* for more information.

4.4. Registration, Admission and Status (RAS) Information

In System Release 5.x with the NetTSC embedded stack, registration functionality was configurable by setting parameter values in the *config.val* file. Functionality was provided in stages as described below:

- A setting determined if working with RAS protocol was enabled or disabled

- If working with the RAS protocol was enabled, then another setting determined if automatic registration at initialization time was enabled or disabled.
- If automatic registration at initialization time was enabled, then other settings determined:
  - Whether working with multicast or unicast is required and the respective address
  - The terminal alias, supported prefixes and time-to-live information (optional)

In System Release 6.0 with the host-based stack, registration is achieved using the **gc_ReqService( )** function. The following rules apply:

- The application must use a valid board device handle that was previously obtained using the **gc_OpenEx( )** function, for example, **gc_OpenEx(&boardDevice,":N_iptB1:P_IP",EV_ASYNC, NULL)** .
- The application must perform initial discovery and registration before handling any calls.
- When the application is registered and has active calls, any deregistration or switching to a different gatekeeper must be done when all line devices in the system are in the Idle state.

The **gc_ReqService( )** function is used for both registration and deregistration and includes a GC_PARM_BLK that can be populated with parameters that determine the following:

- The operation to be performed (register or deregister)
- The sub-operation to be performed (set registration information, add to registration information, delete one component of registration information by value, delete all registeration information)
- Registration address information
- Any local aliases to be included in the registration (when the registration target is H.323 only)
- Any supported prefixes to be included in the registration (when the registration target is H.323 only)

4.4.1. Registration

The **gc_ReqService( )** function is used for registration. Registration information is stored locally and can be retained or discarded when deregistering. IP registration address information is stored in the IP_REGISTER_ADDRESS structure that has the following definition:

```
typedef struct
{
    char    reg_client[IP_REG_CLIENT_ADDR_LENGTH];
    char    reg_server[IP_REG_SERVER_ADDR_LENGTH];
    int     time_to_live;
    int     max_hops;
}IP_REGISTER_ADDRESS;
```

Each field has the following meaning:

- **reg_client**: 128 character local address of registering host
- **reg_server**: 64 character local address of gatekeeper
- **time_to_live**: unicast TTL in seconds
- **max_hops**: multicast TTL in seconds

Important defines in this context are:

```
#define IP_REG_MULTICAST_DEFAULT_ADDR "0.0.0.0" /* default multicast
                                                   registration address */
#define IP_REG_SERVER_ADDR_LENGTH   64    /* server address length in characters */
#define IP_REG_CLIENT_ADDR_LENGTH  128    /* client address length in characters */
```

The following paragraphs contain code segments that demonstrate how to populate the GC_PARM_BLK with the parameters required for registration with a gatekeeper.

1. Including two mandatory parameters. The generic Global Call service request feature requires two mandatory parameters for all service requests including registration. The parameters can be included in the GC_PARM_BLK as follows:

```
GC_PARM_BLKP pParmBlock = NULL;
int frc = GC_SUCCESS;
```

```
/****** Two (mandatory) elements that are not related directly to
the server-client negotiation ********/
frc = gc_util_insert_parm_val(&pParmBlock,
                              GCSET_SERVREQ,
                              PARM_REQTYPE,
                              sizeof(char),
                              IP_REQTYPE_REGISTRATION);

frc = gc_util_insert_parm_val(&pParmBlock,
                              GCSET_SERVREQ,
                              PARM_ACK,
                              sizeof(char),
                              1);
```

2. Since Global Call in System Release 6.0 supports both H.323 and SIP protocols, the protocol must be specified. This is achieved by including the following set ID and parameter ID in the GC_PARM_BLK:

```
/******Setting the protocol target***********/
frc = gc_util_insert_parm_val(&pParmBlock,
                              IPSET_PROTOCOL,
                              IPPARM_PROTOCOL_BITMASK,
                              sizeof(char),
                              IP_PROTOCOL_H323); /*can be H323, SIP or Both*/
```

3. The operation (registration) and the sub-operation (set registration information) can be specified by including the following set ID and parameter IDs in the GC_PARM_BLK:

```
/****** Setting the operation to perform ***********/
frc = gc_util_insert_parm_val(&pParmBlock,
                              IPSET_REG_INFO,
                              IPPARM_OPERATION_REGISTER, /* can be Register
                                                            or Deregister */
                              sizeof(char),
                              IP_REG_SET_INFO);  /* can be other relevant
                                                    "sub" operations */
```

4. Registration address information is specified by including the following set ID and parm ID in the GC_PARM_BLK:

```
/****** Setting address information ***********/
IP_REGISTER_ADDRESS registerAddress;
strcpy(registerAddress.reg_server,"101.102.103.104"); /* set server address*/
strcpy(registerAddress.reg_client,"10.20.30.40");     /* set client (self)
                                                         address */
registerAddress.max_hops = regMulticastHops;
registerAddress.time_to_live = regUnicastTTL;

frc = gc_util_insert_parm_ref(&pParmBlock,
                              IPSET_REG_INFO,
                              IPPARM_REG_ADDRESS,
                              (UINT8)sizeof(IP_REGISTER_ADDRESS),
                              &registerAddress);
```

5. An e-mail alias can be specified by including the following set ID and parameter ID in the GC_PARM_BLK. Other alias types can be included by using the parameter IDs: IPPARM_ADDRESS_DOT_NOTATION (for IP addresses), IPPARM_ADDRESS_H323_ID (for H.323 IDs), IPPARM_ADDRESS_PHONE (for phone numbers), IPPARM_ADDRESS_URL (for universal resource locators), and IPPARM_ADDRESS_TRANSPARENT (for unknown types).

```
/**** Setting terminalAlias information  ****/
/**** With H.323 - may repeat this line with different aliases and
        alias types ****/
/**** SIP does not allow the setting of this parm block ****/
frc = gc_util_insert_parm_ref(&pParmBlock,
                              IPSET_LOCAL_ALIAS,
                              (unsigned short)IPPARM_ADDRESS_EMAIL,
                              (UINT8)(strlen("someone@someplace.com")+1),
                               "someone@someplace.com");
```

6. A phone number supported prefix can be specified by including the following set ID and parameter ID in the GC_PARM_BLK. Other supported prefix types can be included by using the parameter IDs: IPPARM_ADDRESS_DOT_NOTATION (for IP addresses), IPPARM_ADDRESS_H323_ID (for H.323 IDs), IPPARM_ADDRESS_PHONE (for phone numbers), IPPARM_ADDRESS_URL (for universal resource locators), and IPPARM_ADDRESS_TRANSPARENT (for unknown types).

```
/**** Setting supportedPrefixes information  ***********/
/**** With H.323 - may repeat this line with different supportedPrefixes and
        supported prefix types ****/
/**** SIP does not allow the setting of this parm block ****/
frc = gc_util_insert_parm_ref(&pParmBlock,
                              IPSET_SUPPORTED_PREFIXES,
                              (unsigned short)IPPARM_ADDRESS_PHONE,
                              (UINT8)(strlen("011972")+1),
                              "011972");
```

7. Once the GC_PARM_BLK has been set up, the **gc_ReqService( )** function can be used to send the request to the gatekeeper.

```
/****** Send the request  ***********/
    unsigned long   serviceID ;
int rc = gc_ReqService(GCTGT_CCLIB_NETIF,
                       boarddev,
                       &serviceID,
                       pParmBlock,
                       NULL,
                       EV_ASYNC);
```

```
      if (rc != GC_SUCCESS)
      {
          printf("failed in gc_ReqService\n");
          return GC_ERROR;
      }
```

4.4.2. Getting Notification of Registration Status

When using the Global Call API, getting notification of registration status is a two step process:

- Detecting a registration event on the board device
- Extracting the status from the GC_PARM_BLK associated with the event

The following code demonstrates how to detect the event on the board device.

```
int rasProcessEvent(METAEVENT *metaevt)
{
    long                type = sr_getevttype();
    LINEDEV             device = sr_getevtdev();
    EXTENSIONEVTBLK*    pextensionBlk = NULL;
    GC_PARM_BLKP        gcParmBlk;
    int                 rc;
    printf("Got event [0x%x] for board device [%d]\n",type,device);
    if(GCEV_SERVICERESP == metaevt->evttype)
    {
        pextensionBlk = (EXTENSIONEVTBLK*)(metaevt->extevtdatap);
        if (NULL == pextensionBlk)
        {
            printf("rasProcessEvent: (NULL == pextensionBlk)\n");
            return FUNCFAIL;
        }

        gcParmBlk  = (&(pextensionBlk->parmblk));

        rc = getExtension(gcParmBlk);
        if (FUNCSUCCESS != rc)
        {
            printf("rasProcessEvent: No InfoElement on Extension Buffer\n");
            return FUNCFAIL;
        }
    }
    return FUNCSUCCESS;
}
```

The following code shows how to extract registration status (accepted or rejected) from the GC_PARM_BLK associated with the event.

```
int getExtension(GC_PARM_BLKP parm_blk)
{
    GC_PARM_DATA        *parmp = NULL;
    static int xxx=0;
    parmp = gc_util_next_parm(parm_blk,parmp);
    if (!parmp)
    {
        return FUNCFAIL;
    }

    while (NULL != parmp)
    {
        xxx++;
        printf("param # = %d\n",xxx);
        switch (parmp->set_ID)
        {
            case IPSET_REG_INFO:
                if  (IPPARM_REG_STATUS == parmp->parm_ID)
                {
                    int value = (*(int *)(parmp->value_buf));
                    switch (value)
                    {
                        case IP_REG_CONFIRMED:
                            switch (g_waitForCmplt)
                            {
                                case REGESTER:
                                    printf("\tGot REG_STATUS - IP_REG_CONFIRMED for 
Regesteration\n");
                                    break;
                                case UNREGESTER:
                                    printf("\tGot REG_STATUS - IP_REG_CONFIRMED for un 
regester\n");
                                    ApplicationExit();
                                    break;
                                default:
                                    break;
                             }
                        case IP_REG_REJECTED:
                            switch (g_waitForCmplt)
                            {
                                case REGESTER:
                                    printf("\tGot REG_STATUS - IP_REG_REJECTED for 
Regesteration\n");
                                    break;
                                case UNREGESTER:
                                    printf("\tGot REG_STATUS - IP_REG_REJECTED for un 
regester\n");
                                    break;
                                default:
                                    break;
                             }
                        default:
                            break;
                    }
                }
                break;
            default:
                printf("\tGot unknown extension setID %d\n",parmp->set_ID);
                break;
        }/* end switch (parmp->set_ID) */

        parmp = gc_util_next_parm(parm_blk,parmp);
    }
    return FUNCSUCCESS;
}
```

4.4.3. Sending Nonstandard Registration Messages

Non-standard registration messages are sent to the gatekeeper using the **gc_Extension( )** function with an extension ID (**ext_ID**) of IPEXTID_SENDMSG. A GC_PARM_BLK associated with the **gc_Extension( )** function contains the required parameters and the message data. The following code demonstrates how to set up a GC_PARM_BLK and use the **gc_Extension( )** function to send a nonstandard message to the gatekeeper.

```
int sendNonStandardRasMsg()
{
    GC_PARM_BLKP    gcParmBlk = NULL;
    int frc;

    /******Setting the protocol target******/
    frc = gc_util_insert_parm_val(&pParmBlock,
                                  IPSET_PROTOCOL,
                                  IPPARM_PROTOCOL_BITMASK,
                                  sizeof(char),
                                  IP_PROTOCOL_H323); /* can only be H323 target /*
                                                     /* for this task */

    frc = gc_util_insert_parm_val(&gcParmBlk,
                                  IPSET_MSG_REGISTRATION,
                                  IPPARM_MSGTYPE,
                                  sizeof(int),
                                  IP_MSGTYPE_REG_NONSTD);

    frc = gc_util_insert_parm_ref(&gcParmBlk,
                                  IPSET_NONSTANDARDDATA,
                                  IPPARM_NONSTANDARDDATA_OBJID,
                              (unsigned char)(strlen(Boards[1].RegData.NonStdObjID) +1),
                                  (void*)Boards[1].RegData.NonStdObjID);

    frc = gc_util_insert_parm_ref(&gcParmBlk,
                                  IPSET_NONSTANDARDDATA,
                                  IPPARM_NONSTANDARDDATA_DATA,
                                  (unsigned
char)(strlen(Boards[1].RegData.NonStdCmd)+1),
                                  (void*)(Boards[1].RegData.NonStdCmd) );

    if (gc_Extension(GCTGT_CCLIB_NETIF,
                     Boards[1].device,
                     IPEXTID_SENDMSG,
                     gcParmBlk,
                     NULL,
                     EV_ASYNC)<0)
    {
        printf("gc_Extention failed");
    }

    gc_util_delete_parm_blk(gcParmBlk);
    return FUNCSUCCESS;
}
```

4.4.4. Deregistration

Deregistration is achieved using the **gc_ReqService( )** by specifying the IPSET-REG_INFO set ID and the IPPARM_OPERATION_DEREGISTER parameter ID in the GC_PARM_BLK associated with the **gc_ReqService( )** function. In addition, the application can choose to keep the registration information that is stored locally or to discard it by setting the IPPARM_OPERATION_DEREGISTER parameter to a value of IP_REG_MAINTAIN_LOCAL_INFO (deregister but keep registration information locally) or IP_REG_DELETE_ALL (deregister and discard the registration information stored locally).

The following code example shows how to deregister from a gatekeeper and discard the registration information stored locally.

```
int unregister()
{
    GC_PARM_BLKP         pParmBlock = NULL;
    unsigned long        serviceID = 1;
    int                  rc;

    /******Setting the protocol target******/
    rc = gc_util_insert_parm_val(&pParmBlock,
                                 IPSET_PROTOCOL,
                                 IPPARM_PROTOCOL_BITMASK,
                                 sizeof(char),
                                 IP_PROTOCOL_H323); /* can be H323 or SIP /*

    rc = gc_util_insert_parm_val(&pParmBlock,
                                 IPSET_REG_INFO,
                                 IPPARM_OPERATION_DEREGISTER,
                                 sizeof(char),
                                 IP_REG_DELETE_ALL);

    rc = gc_ReqService(GCTGT_CCLIB_NETIF,
                       Boards[1].device,
                       &serviceID,
                       pParmBlock,
                       NULL,
                       EV_ASYNC);

    if ( GC_SUCCESS != rc)
    {
        printf("gc_ReqService failed while unregestering\n");
        gc_util_delete_parm_blk(pParmBlock);
        return FUNCFAIL;
    }
    g_waitForCmplt = UNREGESTER;
    printf("Unregester request to the GK was sent ...\n");
    printf("the application will not be able to make calls !!! so it will EXIT\n");
    gc_util_delete_parm_blk(pParmBlock);
    return FUNCSUCCESS;
}
```

4.5. Setting Coder Information

In System Release 5.x, coder information could be specified in the *config.val* file. In System Release 6.0, coder information can only be set at run time using the Global Call API. Coder information can be set:

- On a system-wide basis using **gc_SetConfigData( )**
- On a per line device basis using **gc_SetUserInfo( )**
- On a per call device using **gc_MakeCall( )**

Any coder information that is set using **gc_SetUserInfo( )** overrides coder information set using **gc_SetConfigData( )** and in turn any coder information set using **gc_MakeCall( )** overrides coder information set using **gc_SetUserInfo( )**. The coder information is specified in a GC_PARM_BLK associated with the respective function. The set ID is GCSET_CHAN_CAPABILITY and the parameter ID is IPPARM_LOCAL_CAPABILITY which is a structure of type IP_CAPABILITY that contains the coder details.

See the *Global Call IP Technology User's Guide* for information on the supported coders and how to set them.

5. Compatibility Issues to be Fixed in Future Releases

The following are known compatibility issues that are expected to be fixed in future releases:

- In the host-based stack implementation, **gc_OpenEx( )** and **gc_ReleaseCallEx( )** are supported in both synchronous and asynchronous mode. All other functions are supported in asynchronous mode only.
- The host-based stack implementation does not currently support RTCP Reports.
- The **gc_GetCTInfo( )** function, which is supported in the NetTSC embedded stack implementation is not currently supported in the host-based stack implementation.

6. Compatibility Issues

The following compatibility issues necessitate changes to applications developed using the NetTSC embedded stack implementation in order to run on the host-based stack implementation in System Release 6.0:

- The *config.val* file is no longer being used. Parameters previously configurable in *config.val* are now configurable using Global Call API functions. See *Chapter 4. config.val File Redundancy* for more information.
- In the embedded stack implementation, when using **gc_OpenEx( )**, the protocol identifier in the **numberstr** parameter is P_H323_R and each IP Media device (ipmBxCy) is bound to the corresponding IP network device (iptBxTy), that is BxCy = BxTy. In the host-based stack implementation, the protocol identifier is P_H323 and the IP Media device is not bound to a corresponding IP network device.
- The **gc_SetConfigData( )** function uses different target types and target IDs in the different implementations as follows:
  - In the NetTSC embedded stack implementation, the target type is GCTGT_PROTOCOL_SYSTEM and the target ID (protocol ID) had to be retrieved using the **gc_QueryConfigData( )** function.
  - In the host-based stack implementation, the target type is GCTGT_CCLIB_NETIF and the target ID is the board device ID.

  In addition, parameter settings apply not only to newly opened devices but also to devices already opened. See the *Global Call IP Technology User's Guide* for more information.
- In the NetTSC embedded stack implementation, the GCEV_EXTENSION event had a dual role acting as the completion event for the **gc_Extension( )** function and as an unsolicited event. In the host-based stack implementation, the completion event for **gc_Extension( )** is GCEV_EXTENSIONCMPLT. The GCEV_EXTENSION event is reserved for unsolicited events only.
- In the NetTSC embedded stack implementation, the number of IP network devices allocated depended on the number and type of Intel® NetStructure™ DM/IP boards in the system. In the host-based stack implementation, the number of IP network devices that will be allocated is configurable using the

IPCCLIB_START_DATA structure associated with the **gc_Start( )** function. See *Section 4.1. Setting Stack Allocation Parameters* for more information.

- In System Release 6.0, the usage of IPPARM_UII_ALPHANUMERIC is different than in the NetTSC embedded stack implementation. See *Section 6.4. UII Alphanumeric Parameter Usage* below for more detail.

6.1. Retrievable Call Information Compatibility

Table 2 shows the retrievable call information in the host-based stack implementation in System Release 6.0 and indicates the level of compatibility with the embedded stack implementation in System Release 5.x.

Table 2. Retrievable Call Information and Backward Compatibility

Parameter	**Set ID**	**Parameter ID**	**System Release 6.0 Backwards Compatible**
RTCP information	IPSET_ CALLINFO	IPPARM_ RTCPINFO	Not supported in System Release 6.0.
Coder information	GCSET_CHAN_ CAPABILITY	IPPARM_ LOCAL_ CAPABILITY	No, see *Section 6.3. Coder Information and Backward Compatibility*.
Display Information	IPSET_ CALLINFO	IPPARM_ DISPLAY	Yes
User-to-User Information	IPSET_ CALLINFO	IPPARM_ USERUSER_ INFO	Yes
Call Duration	IPSET_ CALLINFO	IPPARM_ CALLDURATION	Yes

Parameter	**Set ID**	**Parameter ID**	**System Release 6.0 Backwards Compatible**
Phone List	IPSET_ CALLINFO	IPPARM_ PHONELIST	Yes
Nonstandard Object ID	IPSET_ CALLINFO	IPPARM_NON STANDARDDATA_ OBJID	No, see *Section 6.2. Nonstandar d Object ID and Nonstandard Data Backward Compatibility*.
Nonstandard Data	IPSET_ CALLINFO	IPPARM_NON STANDARDDATA_ DATA	No, see *Section 6.2. Nonstandar d Object ID and Nonstandard Data Backward Compatibility*.
Vendor Product ID	IPSET_ VENDORINFO	IPPARM_ VENDOR_ PRODUCT_ID	Yes
Vendor Version ID	IPSET_ VENDORINFO	IPPARM_ VENDOR_ VERSION_ID	Yes
H.221 Nonstandard information	IPSET_ VENDORINFO	IPPARM_ H221NONSTD	Yes
Conference ID	IPSET_ CONFERENCE	IPPARM_ CONFERENCE_ID	Yes
Conference Goal	IPSET_ CONFERENCE	IPPARM_ CONFERENCE_ GOAL	Yes

Parameter	**Set ID**	**Parameter ID**	**System Release 6.0 Backwards Compatible**
Call ID	IPSET_ CALLINFO	IPPARM_ CALLID	Not supported in System Release 5.x.

6.2. Nonstandard Object ID and Nonstandard Data Backward Compatibility

In System Release 5.x, the Nonstandard Object ID and Nonstandard Data parameters used the IPSET_CALLINFO parameter set ID. Since these parameters are not call information parameters, they have been changed to use a more logically named IPSET_NONSTANDARDCONTROL parameter set ID. IPSET_NONSTANDARDCONTROL uses the same nonstandard parameter IDs as previously used by IPSET_CALLINFO.

In addition, IPSET_NONSTANDARDCONTROL also supports the IPPARM_H221NONSTANDARD parameter ID.

6.3. Coder Information and Backward Compatibility

The GCEV_EXTENSION event with a parameter set ID of IPSET_MEDIA_STATE is used to signal that media has either started or stopped streaming in the transmit or the receive directions:

- If the parameter is IPPARM_TX_CONNECTED, then media has begun to stream in the transmit direction.
- If the parameter is IPPARM_RX_CONNECTED, then media has begun to stream in the receive direction.

The coder configuration is contained in the event data as an IP_CAPABILITY structure.

6.4. UII Alphanumeric Parameter Usage

In the NetTSC embedded stack implementation, the IPPARM_UII_ALPHANUMERIC parameter had a dual role:

- If the application was using out-of-band (OOB) DTMF (as defined by setting **PrmDTMFXferMode** to 2 [default] in the *.config* file), then IPPARM_UII_ALPHANUMERIC was used for DTMF.
- If the application was using in-band DTMF (as defined by setting **PrmDTMFXferMode** to 1 in the *.config* file), then IPPARM_UII_ALPHANUMERIC was used to send messages to the application at the remote side.

In the host-based stack implementation, the *.config* file is not used. DTMF and application data transfer is achieved as follows:

- The application uses IPSET_DTMF with IPPARM_DTMF_ALPHANUMERIC (which is the same value as IPPARM_UII_ALPHANUMERIC) to send DTMF.
- The application can use User Input Indication (UII) non-standard, or Q931 User-to-User Information (UUI) to send information to an application at the remote end.

See the *Global Call IP Technology User's Guide* for more information.

7. Enhanced Functionality

The host-based stack implementation in System Release 6.0 includes the following enhancements over the NetTSC embedded stack implementation:

- Support for both H.323 and SIP protocols
- Support for RFC2833.
- Support for T.38 fax.
- In the host-based stack implementation, both Transmit and Receive coders can be specified. In the NetTSC embedded stack implementation, only Transmit coders could be specified.
- The behavior of the **gc_SetUserInfo( )** with the GC_SINGLECALL (per call basis) and GC_ALLCALLS (per device basis) options is different as follows:
    - In the NetTSC embedded stack implementation, only coder information could be set using GC_ALLCALLS (per line device basis); all other information could be set using GC_SINGLECALL only (per call basis).
    - In the host-based stack implementation, all parameters (coder info, conference goal, connection method, display information, nonstandard data, nonstandard control, phone list, tunneling, and user-to-user information) can be set using both GC_SINGLECALL and GC_ALLCALLS.
- In the NetTSC embedded stack implementation, the Proceeding message was sent automatically by the stack. In the host-based stack implementation, the application can determine if the Proceeding message should be sent manually or automatically by the stack. See *Section 7.1. Specifying Automatic or Manual Sending of the Proceeding Message* below for more detail.
- In the NetTSC embedded stack implementation, routing using **gc_Listen( )**, **gc_UnListen( )** had to be done in synchronous mode, but these functions were non-blocking. In the host-based stack implementation, it is recommended to use **gc_Listen( )** and **gc_Unlisten( )** in asynchronous mode. Using **gc_Listen( )** and **gc_UnListen( )** in synchronous mode is supported, but these functions are blocking in the host-based implementation.

7.1. Specifying Automatic or Manual Sending of the Proceeding Message

In the NetTSC embedded stack implementation, during call setup, the Proceeding message was sent automatically. In the host based stack implementation, the application can configure if the Proceeding message should be sent manually by the application (using the **gc_CallAck( )** function) or automatically by the stack.

This configuration is done using the **gc_SetConfigData( )** function with the following set ID and parameter ID in the associated GC_PARM_BLK:

- GCSET_CALL_CONFIG
    - GCPARM_CALLPROC - Possible values are:
        - GCCONTROL_APP - The Proceeding message should be sent by the application
        - GCCONTROL_TCCL - The Proceeding message is sent automatically by the stack

The default is that the Proceeding message is sent manually by the application by issuing **gc_CallAck( )**. If the automatic option is selected, there may be performance issues at the receiving side.