Global Call E1/T1 CAS/R2 Technology User's Guide

for Linux and Windows

05-0615-011

Publication Date: November 2003

Document Number: 05-0615-011

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

For **Sales Offices** and other contact information, visit the Where to Buy Intel Telecom Products at:
http://www.intel.com/buy/wtb/wtb1028.htm

Table of Contents

List of Tables

1. How to Use This Guide

This guide is for users who use the Global Call application programming interface (API), the Global Call Protocol Development Kit Run Time (PDKRT) call control library, or the Global Call Interface Control API (ICAPI) call control library and related software to develop Linux or Windows applications in an E1 channel associated signaling (CAS) or T1 robbed bit environment.

Complete reference information and programming guidelines for the Global Call API are given in the companion documents, *Global Call API Library Reference* and *Global Call API Programming Guide*. Certain Global Call functions have additional functionality or perform differently when used in an E1 CAS or T1 robbed bit environment. The general function descriptions in the *Global Call API Library Reference* do not contain detailed information on a particular technology. Detailed information in terms of the additional functionality or the difference in performance of those functions in an E1 CAS or T1 robbed bit environment is contained in this *Global Call E1/T1 CAS/R2 Technology User's Guide*.

NOTE: Differences between the implementation of a Global Call application in a Linux or a Windows environment are either described parenthetically or are presented in separate paragraphs or sections. Information that is specific to the use of DM3 or Springware boards is identified explicitly. Information that is specific to the use of the PDKRT or ICAPI call control library is identified explicitly. Information about PDK protocols is applicable to DM3 and Springware boards unless otherwise noted. Information about ICAPI protocols is applicable to Springware boards only.

The rest of this chapter describes the organization of this guide, the products covered by this guide, and related publications.

1.1. Organization of this Guide

The information in this guide is organized as follows:

Chapter 2. Signaling Concepts provides an overview of E1 CAS and T1 robbed bit signaling concepts.

Chapter 3. Developing Global Call E1 CAS or T1 Robbed Bit Applications presents guidelines for developing E1 CAS or T1 robbed bit applications.

Chapter 4. Applying Global Call Functions to E1 CAS or T1 Robbed Bit Applications describes the additional functionality or the difference in performance of specific Global Call functions when used in an E1 CAS or T1 robbed bit environment.

Chapter 5. Resource Allocation and Routing describes using dedicated or shared voice resources in an E1 CAS or T1 robbed bit environment.

Chapter 6. Protocols describes the protocol conventions used and programming considerations when incorporating individual country protocol(s) into your application.

Chapter 7. Debugging Applications describes the diagnostic tools available for debugging a Global Call application.

1.2. Intel® Dialogic® Products That Support E1/T1 Signaling

The Global Call software provides a consistent interface across Intel® Dialogic® products interfaced to various networks (for example, E1 CAS, T1 robbed bit, E1 ISDN, T1 ISDN, analog, SS7, and IP). See the Release Guide for your Intel® Dialogic® system release for the Intel® Dialogic® product combinations that provide E1 CAS signaling and T1 robbed bit signaling capabilities.

1.3. Related Information

Use this guide in conjunction with the following manuals:

- *Global Call API Library Reference* – provides a reference to all functions, events, data structures, and error codes in the Global Call API library.
- *Global Call API Programming Guide* – provides guidelines for developing applications using the Global Call API.

- *Global Call Country Dependent Parameters (CDP) Configuration Guide* – describes the parameters associated with each of the countries needed for utilizing Global Call.
- Release Notes for the Global Call Protocols package – provides information about installing and using Global Call protocols, including how to configure DM3 boards to run PDK protocols.

The following information is also useful:

- Release Guide for your system release – provides information about the system release, system requirements, software and hardware features, supported hardware, and release documentation.
- Release Update for your system release (available on the Technical Support Web site only) – describes compatibility issues, restrictions and limitations, known problems, and late-breaking updates or corrections to the release documentation. The Release Update is updated with new information as needed during the lifecycle of the release.
- *http://developer.intel.com/design/telecom/support* – Technical Support Web site which contains developer support information, downloads, release documentation, technical notes, application notes, a user discussion forum, and more.

For additional information about E1 or T1 telephony, see the following publications:

- R2 MF Signaling References
  - *Specifications of Signaling Systems R1 and R2*, International Telegraph and Telephone Consultative Committee (CCITT), Blue Book Vol. VI, Fascicle VI.4, ISBN 92-61-03481-0
  - *General Recommendations on Telephone Switching and Signaling*, International Telegraph and Telephone Consultative Committee (CCITT), Blue Book Vol. VI, Fascicle VI.1, ISBN 92-61-03451-9
- T1 Robbed Bit Signaling References
  - Bellamy, John, *Digital Telephony*, 2nd ed. New York: John Wiley & Sons, 1991

- Fike, John L., and George Friend, *Understanding Telephone Electronics*, Indiana: Howard W. Sams & Company, 1988
- Flanagan, William A., *The Guide to T-1 Networking*, 4th ed. New York, Telecom Library Inc., 1990
- *LATA Switching Systems Generic Requirements (LSSGR)*, Bellcore Technical Reference TR-TSY-000064, Issue 2, July 1987, and modules, Bellcore

2. Signaling Concepts

This chapter provides an overview of how digits and signaling information are transmitted when a telephone call is made, and describes the T1 robbed bit, E1 CAS, R2 MF, and Direct Dialing In (DDI) signaling concepts.

2.1. Making Telephone Calls: Transmission of Digits and Signaling Information

Historically, making a telephone call started with taking your telephone handset out of its cradle. This action caused your telephone to go off-hook. For analog telephones, going off-hook closes a circuit (called the local loop) connected to the local Central Office (CO) and causes a loop current to flow through the local loop circuit created.

The CO reacts by generating dial tone (typically, a combination of 350 Hz and 440 Hz tones), which indicates that you can dial. Traditionally, you would dial your number using pulse dialing (also called rotary dialing). Pulse dialing sends digit information to the CO by momentarily opening and closing (or breaking) the local loop from the calling party to the CO. This local loop is broken once for the digit 1, twice for 2, etc., and 10 times for the digit 0. As each number is dialed, the loop current is switched on and off, resulting in a number of pulses being sent to your local CO.

Alternatively, you may dial a number using tone dialing, wherein sounds represent the digits dialed (0 through 9, # and * are dialing digits). Each digit is assigned a unique pair of frequencies called Dual Tone Multi Frequency (DTMF) digits (see *Table 1. Signaling Used to Dial*). Although DTMF signaling is designed for operation on international networks with 15 multifrequency combinations in each direction, in national networks it can be used with a reduced number of signaling frequencies (for example, 10 multifrequency combinations).

In addition to the DTMF digit standard, telcos also use a Multi Frequency (MF) digit standard (see *Table 1. Signaling Used to Dial*). MF digits are typically used for CO-to-CO signaling. The MF digit standard is similar to the DTMF digit standard except that different pairs of frequencies are assigned. Some MF digits use approximately the same frequencies as DTMF digits; for example, the digit 4

uses 770 and 1209 Hz for DTMF transmissions or 700 and 1300 Hz for MF transmissions. Because of this frequency overlap, MF digits could be mistaken for DTMF digits if the incorrect tone detection is enabled. The accuracy of digit detection depends on:

- the digit sent
- the type of detection, MF or DTMF, enabled when the digit is detected. See the *Voice API Library Reference* for details.

Table 1. Signaling Used to Dial (Hz)

Code	Pulse	DTMF	MF	R2 MF Forward	R2 MF Backward
1	1	697, 1209	700, 900	1380, 1500	1140, 1020
2	2	697, 1336	700, 1100	1380, 1620	1140, 900
3	3	697, 1477	900, 1100	1500, 1620	1020, 900
4	4	770, 1209	700, 1300	1380, 1740	1140, 780
5	5	770, 1336	900, 1300	1500, 1740	1020, 780
6	6	770, 1477	1100, 1300	1620, 1740	900, 780
7	7	852, 1209	700, 1500	1380, 1860	1140, 660
8	8	852, 1336	900, 1500	1500, 1860	1020, 660
9	9	852, 1477	1100, 1500	1620, 1860	900, 660
0	10	941, 1336	1300, 1500	1740, 1860	780, 660
*	-	941, 1209	1100, 1700	1380, 1980	1140, 540
#	-	941, 1477	1500, 1700	1500, 1980	1020, 540

For each call, signaling information (off-hook, number dialed) must be detected by the local CO and then sent to each successive CO until the destination CO is reached. The destination CO attempts to connect to the called party. Concurrently, the destination CO sends back signaling information (such as line busy, network busy signals, etc.) representing the condition or status of the called party's line.

This signaling information passes through the network as audio tones or as signaling bits. The number of tones used and the frequency combinations used to convey this signaling information vary from country to country and from telco to telco. In addition, private networks may combine various signaling techniques.

After dialing, you listen to hear the progress and status of the call:

- Ringing tones (ringback) indicate that ring voltage has been applied to the called party's line.
- A busy tone is heard when the called party's telephone is off-hook.
- A fast busy tone may be heard if the telephone network is busy.
- An operator intercept signal is heard if an invalid number is dialed. The operator intercept signal is three rising tones followed by a recording.

NOTE: No ringing tones are heard when connected to some telcos.

The CO typically indicates the progress of making a call by generating these various tones. When making long distance calls, the telco may make brief drops in loop current to indicate:

- an acknowledgment that the distant CO was reached
- that the calling party's line went off-hook

After a call is connected, a telco service may be requested by a flash-hook. A flash-hook puts the telephone on-hook briefly, long enough for the CO to detect the flash-hook, but not long enough to cause a disconnect. A flash-hook may signal a request for a second dial tone to allow 3-way conferencing or to transfer the call.

At the completion of the call, one or both parties hang up the telephone. Typically, the CO sends a disconnect signal. However, some telcos don't send a disconnect signal; therefore a local CO must use other methods to detect a remote disconnect.

2.1.1. Making Long Distance and Global Telephone Calls

Long distance calls may involve transmitting dialing and other signaling information from the local CO, through several intermediate COs, to the distant called party's CO and then connecting to the called party. A mixture of signaling

systems and protocols may be encountered especially when making global calls. Local call signaling must be translated into signaling that may pass over analog lines, T1 digital trunks, E1 digital trunks, optical fiber, satellite links, etc. All signaling sent over digital trunks must be converted to bits that can be transmitted or multiplexed with the digitized voice transmissions.

Each telco, country, or region tends to apply different signaling standards that must be observed to ensure that a call gets switched through to the called party. For example, some telcos may encode E&M (Ear and Mouth) signals onto the voice path using a single frequency (SF) tone. When present, this tone indicates an on-hook condition. Otherwise, the line is considered to be off-hook (absence of tone). Typically, when the same manufacturer's product is connected to both ends of a digital trunk, then the signaling technique used is transparent as long as all signaling is handled.

2.2. T1 Robbed Bit Signaling Concepts

A T1 trunk operates at 1.544 Mbps divided into 24 time slots with each time slot operating at 64 kbps [digital signal level 1 (DS-1) rate]. A single 8-bit sample from each of 24 voice channels comprises a D4 frame of 24 time slots on a T1 trunk. Twelve D4 frames make up a D4 superframe.

Signaling information is carried on a T1 trunk by two signaling bits, an A-bit and a B-bit. Each time slot in the sixth frame of a D4 superframe has the least significant bit replaced with A-bit signaling information. Likewise, each time slot in the twelfth frame of the D4 superframe has the least significant bit replaced with B-bit signaling information. This method of replacing the least significant bit with signaling information is called robbed bit signaling. Thus, a T1 robbed bit trunk carries all signaling within the voice time slot (channel) itself.

Dialing, if not done using DTMF or MF tones, is accomplished by alternating the A and B signaling bits between 0 and 1 to mimic rotary dial pulses. Signaling bits represent the state of the M lead on the E&M interface of the calling party. When the called party answers, the M lead returns continuous 1s. When a party hangs up, their signal bits revert to 0s to indicate on-hook. Some telcos invert these signaling bits so that 0 = off-hook and 1 = on-hook.

New telco services may require the use of more than the four signaling states provided by the A and B bits. An extended superframe (ESF) adopted by AT&T provides two additional signaling bits, the C-bit in frame 18 and the D-bit in frame 24.

2.3. E1 CAS Signaling Concepts

An E1 digital trunk operates at 2.048 Mbps divided into 32 time slots with each time slot operating at 64 kbps. These 32 time slots include:

- 30 time slots available for up to 30 voice calls
- one time slot dedicated to carrying frame synchronization information (time slot 0)
- one time slot dedicated to carrying signaling information (time slot 16)

With this method of signaling, each traffic channel has a dedicated signaling channel, that is, channel associated signaling (CAS). The signaling for a particular traffic circuit is permanently associated with that circuit. For E1 CAS, the signaling channel for each traffic channel is located in time slot 16, which is multiplexed between all 30 traffic channels.

E1 CAS service is available in Europe, Africa, Australia, and in parts of Asia and South America. The Conference des Administrations Europeenes des Postes et Telecommunications (CEPT) defines how a PCM carrier system in E1 areas will be used. In addition, the E1 CAS service may carry national and international signaling bits set in time slot 0:

- The international bit occupies the most significant bit (bit position 7) in time slot 0 of each frame.
- The national bits occupy bit positions 0 through 4 of time slot 0 of every second frame.

For each E1 CAS call, signaling information is sent to the local CO and then to each successive CO until the destination CO is reached. The destination CO attempts to connect to the called party. Concurrently, the destination CO sends back signaling information representing the condition or status of the called party's line. This signaling information passes through the network as audio tones. R2 MF signaling is the international standard for conveying call status using these

audio tones. However, the number of tones used, the frequency combinations used, and the adherence to the R2 standard can vary from country to country.

Also, whenever a call is switched via networks or protocols that do not support full R2 MF signaling, call information may be lost. Although many protocols do not require call analysis because the called party condition is received via R2 tones, when operating in environments where call information may be lost, call progress tones (busy, ringback, SIT tones, etc.) may be useful in determining the condition of a call.

The following paragraphs describe R2 MF signaling as it is used in a full R2 network, global tone detection considerations, and Global Call call analysis capability. The protocols do not require call analysis because the called party condition is received via R2 tones; but you can modify the country dependent parameters (*.cdp*) file so that the protocol can use either call progress tones or call analysis.

2.4. R2 MF Signaling Concepts

R2 MF signaling is an international signaling system on E1 that transmits numeric and other information relating to the called and the calling subscribers' lines. R2 MF signals that control the call setup are referred to as interregister signals.

For each call, whether an inbound or an outbound call, the entity making the call is the "calling party" and the entity receiving the call is the "called party." For an inbound call, the calling party is eventually connected to a central office (CO) that connects to the customer premises equipment (CPE) of the called party. For this inbound call, the CO is referred to as the outgoing register and the CPE as the incoming register. Signals sent from the CO are forward signals; signals sent from the CPE are backward signals. The outgoing register (CO) sends forward interregister signals and receives backward interregister signals. The incoming register (CPE) receives forward interregister signals and sends backward interregister signals.

For an outbound call, the calling party's CPE connects to the CO that switches the outbound call to the called party. For an outbound call, the signaling described above is reversed. That is, signals sent from the CPE are forward signals and signals sent from the CO are backward signals.

In addition, address signals can provide the telephone number of the called party's line. For national traffic, the address signals can also provide the telephone number of a calling party's line for automatic number identification (ANI) applications.

R2 MF signals used for supervisory signaling on the network are called line signals.

For example, a calling party sends the first dialed digits to the local CO. The local CO uses these digits to determine the next CO in the connection chain. The next CO uses these first dialed digits to determine if they are the destination CO or if the call is to be switched to another CO. Eventually, the call reaches the destination CO. At the destination CO, the call is received and acknowledged. The destination CO eventually gets the last dialed digits, which exactly identify the called party.

The destination CO checks the called party's line to determine if it is clear, idle, busy, etc. The destination CO then generates and sends a B-tone backwards to the calling party to indicate the condition of the line. If the called party's line is free, the destination CO applies ringing to the line and sends ringback tones backwards to the calling party. When the called party answers the call, the calling party is switched through to the called party. If the called party's line is busy, or in some other condition, the destination CO sends this information backwards to the calling party via R2 tones. The local CO sends all information received from the destination CO to the calling party. When calls are made in countries that adhere to the full R2 protocol standard (for example, Belgium), the condition of the called party's line is always returned to the calling party.

When traversing networks, protocols, or countries, R2 tonal information can be lost. For example:

- In Italy, for an ICAPI protocol, the calling party would need to use busy tone 103 and ringback tone 105 to determine the condition of the called party's line.
- When the call is switched over a T1 span, the B-tones (also called Group B signals, see *Section 2.4.2. R2 MF Signal Meanings*) are lost and the condition of the called party's line cannot be detected using R2 tones. In this environment, the application must rely on the call progress tones received to determine the condition of the called party's line.

- In Spain, the network is not a full Socotel backbone; therefore, B-tones defining the condition of the called party's line may or may not be sent backwards to the calling party.

2.4.1. R2 MF Multifrequency Combinations

R2 MF signaling uses a multifrequency code system based on six fundamental frequencies in the forward direction (1380, 1500, 1620, 1740, 1860, and 1980 Hz) and a different set of six frequencies in the backward direction (1140, 1020, 900, 780, 660, and 540 Hz).

Each signal is composed of two of the six frequencies, providing 15 different tone combinations in each direction. Although R2 MF signaling is designed for operation on international networks with 15 multifrequency combinations in each direction, in national networks it can be used with a reduced number of signaling frequencies (for example, 10 multifrequency combinations). See the *Voice API Library Reference* for lists of these signal tone pairs.

2.4.2. R2 MF Signal Meanings

The 15 forward signals are classified into Group I forward signals and Group II forward signals. The 15 backward signals are classified into Group A backward signals and Group B backward signals.

In general, Group I forward signals and Group A backward signals are used to control call setup and to transfer address information between the outgoing register (CO) and the incoming register (CPE). The incoming register can signal the outgoing register to change over to Group II and Group B signaling.

Group II forward signals provide the calling party's category and Group B backward signals provide the condition of the called subscriber's line. Group B signals, also called B-tones, are typically the last tone in the protocol. For example, typically a B-3 tone indicates that the called party's line is busy.

Signaling must always begin with a Group I forward signal followed by a Group A backward signal that serves to acknowledge the signal just received; this Group A backward signal may request additional information. Each signal requires a

response from the other party. Each response becomes an acknowledgment of the event and an event to which the other party must respond.

Backward signals serve to indicate certain conditions encountered during call setup or to announce switchover to changed signaling, for example, forward signaling switching over to backward signaling. Changeover to Group II and Group B signaling allows information about the state of the called subscriber's line to be transferred.

The incoming register backward signals can request:

- transmission of address:
    - send next digit
    - send digit previous to last digit
    - send second digit previous to last digit sent
    - send third digit previous to last digit sent
- category of the call (the nature and origin):
    - national or international call
    - operator or subscriber
    - data transmission
    - maintenance or test call
- whether the circuit includes a satellite link
- country code and language for international calls
- information on use of an echo suppresser

The incoming register backward signals can indicate:

- address complete - send category of call
- address complete - put call through
- international, national, or local congestion
- condition of subscriber's line:
    - send SIT to indicate long term unavailability
    - line busy
    - unallocated number

- line free - charge on answer
- line free - no charge on answer (only for special destinations)
- line out of order

The meaning of certain forward multifrequency combinations may also vary depending upon their position in the signaling sequence.

See the *Voice API Library Reference* for more details and definitions of R2 MF signals.

2.4.3. R2 MF Compelled Signaling

Compelled signaling protocols vary from country to country and are grouped into two main categories, both of which are supported by the Global Call software:

- R2 MF derived from the CCITT (International Telegraph and Telephone Consultative Committee) standard, where the response tones can carry information from the receiver to the sender. This standard provides a consistent handshake, where the sender always initiates with a forward tone, and the receiver always responds with a backward tone.
- MF Socotel, where the response tone is a standard, single frequency acknowledgment tone that cannot carry additional information. In this standard, the handshake of forward and backward tones changes direction when the receiver needs to send information back to the sender.

The Global Call software provides network device independence by shielding the application from protocol-specific details while giving access to each protocol's full range of features. The compelled signaling feature uses tone generation and detection IDs that are defined at system initialization.

R2 MF interregister signaling uses forward and backward compelled signaling. With compelled signaling, each signal is sent until a response (a return) signal is generated. This return signal is sent until responded to by the other party. Each signal stays on until the other party responds, thus compelling a response from the other party. Compelled signaling provides a balance between speed and reliability because it adapts its signaling speed to the working conditions with a minimum loss of reliability.

Compelled signaling must always begin with a Group I forward signal. For an inbound call:

- The CO starts to send the first forward signal.
- As soon as the CPE recognizes this signal, the CPE starts to send a backward signal that serves as an acknowledgment and may also request additional information.
- As soon as the CO recognizes the CPE acknowledging signal, the CO stops sending the forward signal.
- As soon as the CPE recognizes the end of the forward signal, the CPE stops sending the backward signal.
- As soon as the CO recognizes that the CPE stopped sending the backward signal, the CO may start to send the next forward signal.

The above scenario describes the CPE handling of an inbound call. The roles of the CO and the CPE are reversed when the CPE makes an outbound call.

2.5. Direct Dialing In (DDI) Service

Since DTMF, MF, and R2 MF tone signals can provide the telephone number of the called subscriber's line, these signals may be used by applications providing Direct Dialing In (DDI) service, also called dialed number identification service (DNIS) and analog DNIS for direct inward dialing (DID).

DDI service allows an outside caller to dial an extension within a company without requiring an operator's assistance to transfer the call. The CO passes the last 2, 3, or 4 digits of the dialed number to the CPE, and the CPE completes the call.

3. Developing Global Call E1 CAS or T1 Robbed Bit Applications

This chapter offers advice and suggestions for programmers designing and coding Global Call E1 CAS or T1 robbed bit applications in a Linux or Windows environment. Topics include the following:

- Global tone detection (GTD) tone considerations
- Call progress and call analysis
- Header files
- Resource association
- Alarm handling
- Run time configuration of the PDKRT call control library and PDK protocol parameters
- Determining the protocol version

For more information about developing applications with Global Call, see the *Global Call API Programming Guide*.

3.1. Global Tone Detection (GTD) Tone Considerations

NOTE: The information in this section is applicable to Springware boards only.

Global Call will delete all tones and load internally required tones (used for call progress) under either of the following circumstances:

- If there is a voice device attached to the network device during **gc_OpenEx( )**
- When **gc_Attach( )** or **gc_AttachResource( )** is called, if at least one of the following statements is true:
    - This is the first time the voice resource is being attached to a network device opened in the PDK library (either implicitly via **gc_OpenEx( )**, or explicitly via **gc_Attach( )** or **gc_AttachResource( )**).

- Downloading of tones is enabled (**gc_SetParm(ldev, GCPR_LOADTONES, GCPV_ENABLE)**).

If Global Call deleted all tones during **gc_OpenEx( )**, **gc_Attach( )**, or **gc_AttachResource( )** as described above, then the application must reload any tones that it has loaded. It is recommended that the application not download tones for a voice device prior to calling **gc_OpenEx( )** if the voice device is specified in the **gc_OpenEx( )**, as the tones will be deleted. Similar considerations apply to **gc_Attach( )** and **gc_AttachResource( )**.

It is the application's responsibility to ensure that the internally required tones are available to the protocol during call setup. This can be done by either:

- Never deleting all tones, or
- If the application has deleted all tones while the voice resource is not attached, enabling downloading of tones

CAUTION

The application must not delete all tones while the voice resource is attached.

In any case, the application may not delete internally required tones during call setup.

NOTE: For PDK and ICAPI protocols, the tone IDs for any additional tones that must be redefined after calling **gc_Attach( )** or **gc_AttachResource( )** cannot be in the range from 101 to 189.

The overhead of downloading tones is expensive. Therefore, for any application that calls **gc_Attach( )** or **gc_AttachResource( )** several times on the same device (for example, when resource sharing), this overhead can be avoided by calling **gc_SetParm(ldev, GCPR_LOADTONES, GCPV_DISABLE)**. This **gc_SetParm( )** function should be called after the first call to the **gc_Attach( )** or **gc_AttachResource( )** function, or after the call to the **gc_OpenEx( )** function if the voice device is specified in **gc_OpenEx( )**. It is then the application's responsibility not to delete all tones on the voice device.

3.2. Call Progress and Call Analysis

Call analysis consists of both pre-connect and post-connect information about the progress of the call. Pre-connect call progress determines the status of the call connection, that is, busy, no dial tone, no ringback, etc. Post-connect call analysis, which is also known as media type detection, determines the destination party's media type, that is, answering machine, fax, voice, etc.

NOTE: In Global Call terminology, the term call analysis is used interchangeably with the term call progress.

The following sections discuss:

- Call analysis with DM3 boards
- Call analysis with Springware boards
- Call analysis functionality for PDK protocols
- Call analysis functionality for ICAPI protocols

3.2.1. Call Analysis with DM3 Boards

NOTE: When using DM3 boards, Global Call provides a consistent method of pre-connect call progress and post-connect call analysis across analog, CAS, and ISDN protocols. Refer to the *Global Call API Programming Guide* for information about this method of call progress analysis.

The information included below is specific to the E1/T1 CAS technology and is provided for backward compatibility only. For new applications, it is recommended to use the cross-technology call progress analysis method described in the *Global Call API Programming Guide*.

There are two methods available for call analysis when using DM3 boards: the Global Call method and the **dx_dial( )** method.

The Global Call media detection method is especially useful for performing post-connect call analysis. When activated by setting the **GCPR_MEDIADETECT** parameter to GCPV_ENABLE for a particular channel, post-connect call analysis is performed as part of the **gc_MakeCall( )** function's operation. The **gc_MakeCall( )** function is used to place a call; the signal detector analyzes the incoming signals to perform call progress analysis.

After the normal **gc_MakeCall( )** processing finishes and GCEV_CONNECTED event is sent, call analysis runs and generates a GCEV_MEDIADETECTED event that tells the application the result of the analysis (for example, FAX, PVD, or PAMD is detected).

The outcome of the analysis determines the events generated and the action that can be taken as follows:

- If the call is successful, **gc_MakeCall( )** finishes and a GCEV_CONNECTED event is sent, call analysis runs, and generates a GCEV_MEDIADETECTED event. The **gc_ResultValue( )** and **gc_GetCallInfo( )** functions can then be used to get more information about the type of media detected, such as voice, answering machine, and fax.
- If the call is not successful—for example, there is no ringback—a GCEV_DISCONNECTED event is generated and the **gc_ResultValue( )** function can be used to retrieve the reason for the failure. See the *Global Call API Library Reference* for error codes and the *gcerr.h* file for more information.

NOTE: The information above applies when using **gc_MakeCall( )** in asynchronous or synchronous mode. However, in synchronous mode, since the **gc_MakeCall( )** function must complete, the GCEV_MEDIADETECTED event is generated after the call is connected.

GCPR_MEDIADETECT and **GCPR_CALLPROGRESS** parameter settings for **gc_SetParm( )** actually allow the application to specify whether pre- or post-connect call analysis or both should be activated. This method for achieving this is shown in *Table 2*.

Table 2. Global Call Call Progress Settings

	GCPR_CALLPROGRESS=GCPV_DISABLE	**GCPR_CALLPROGRESS=GCPV_ENABLE (default)**
GCPR_MEDIADETECT=GCPV_DISABLE (default)	No call progress	Pre-connect call progress **only**

	GCPR_CALLPROGRESS=GCPV_DISABLE	GCPR_CALLPROGRESS=GCPV_ENABLE (default)
GCPR_MEDIADETECT=GCPV_ENABLE	No call progress	Full call progress

As can be seen in this table, the default behavior (**GCPR_MEDIADETECT = GCPV_DISABLE**) disables media detection but actually activates pre-connect call progress for CAS protocols. To enable full call progress analysis, set the **GCPR_MEDIADETECT** parameter to GCPV_ENABLE for the respective channel.

NOTE: For this Global Call media detection to work, a voice device must be attached to the line device and properly routed. Failure to do so will cause subsequent outgoing call attempts to fail.

The **GCPR_CALLPROGRESS** parameter can be used to enable or disable pre-connect call progress. When combined with **GCPR_MEDIADETECT**, this allows the application to specify whether to use pre-connect call progress only or full call progress. If **GCPR_CALLPROGRESS = GCPV_DISABLE**, there will be no call progress at all, regardless of the setting of **GCPR_MEDIADETECT**.

Table 3 explains call analysis support via the Global Call interface. The table applies to DM3 CAS protocols with flexible routing clusters, provided that a voice device is attached to the network device. The table also applies to any fixed routing configuration regardless of the protocol. Check on a protocol-by-protocol basis, as some might not support call analysis at all.

Table 3. Call Analysis Support on DM3 Boards with CAS

Call Analysis Feature	Support on DM3	How Obtained/Notes
Busy	Yes	Upon DISCONNECT event, call **gc_ResultValue( )**.
No ringback	No	

Call Analysis Feature	Support on DM3	How Obtained/Notes
SIT	Yes	Upon DISCONNECT event, call **gc_ResultValue( )**.
No answer	Yes	Upon DISCONNECT event, call **gc_ResultValue( )**.
Cadence break	No	
Discarded	No	
NA	Yes	Use **GCPR_MEDIADETECT** parameter. Upon MEDIADETECTED event, call **gc_GetCallInfo( )**.
Unknown	Yes	Use **GCPR_MEDIADETECT** parameter. Upon MEDIADETECTED event, call **gc_GetCallInfo( )**.
PVD	Yes	Use **GCPR_MEDIADETECT** parameter. Upon MEDIADETECTED event, call **gc_GetCallInfo( )**.
PAMD	Yes	Use **GCPR_MEDIADETECT** parameter. Upon MEDIADETECTED event, call **gc_GetCallInfo( )**.
Fax	Yes	Use **GCPR_MEDIADETECT** parameter. Upon MEDIADETECTED event, call **gc_GetCallInfo( )**.
In progress	Yes	Use **GCPR_MEDIADETECT** parameter. Upon MEDIADETECTED event, call **gc_GetCallInfo( )**.

Note that the call analysis time-out parameters values apply, and they are configurable by the user. (They cannot be changed at runtime.) The parameters are **CaSignalTimeout**, **CaAnswerTimeout**, and **CaPvdTimeout**; their values are found in the CHP section of the configuration (*.config*) file. However, they apply

only to post-connect call analysis and are not used until the call moves from an initiated to a Proceeding, Alerting, or Connected state.

Another option for call analysis is provided by the voice API, which provides post-connect call analysis on DM3 boards through the **dx_dial( )** function. Note that the Global Call method and the **dx_dial( )** method are mutually exclusive, so you must choose one or the other.

3.2.2. Call Analysis with Springware Boards

The **gc_GetCallInfo( )** function is used immediately following the receipt of a GCEV_CONNECTED event to retrieve this post-connect information notifying of the media type of the answering party. See the *Global Call API Library Reference* for more information.

Call analysis tones such as dial tone, ringback, busy, and fax are defined either in the firmware (Global Tone Detection and Global Tone Generation), or in the country dependent parameters (*.cdp*) file, or a combination of both. Tones defined in the firmware can be enabled or disabled by configuring parameters in the DX_CAP (call analysis parameter) data structure. Similarly, the DX_CAP data structure can be used to configure the voice detection algorithm that distinguishes answering machine or human speech. The default parameter values defined in the DX_CAP data structure can be changed by the **gc_LoadDxParm( )** function to fit the needs of your application. For a detailed description of enhanced call analysis (PerfectCall) and how to use call analysis, see the *Voice API Programming Guide*. For a detailed description of the **gc_LoadDxParm( )** function, see the *Global Call API Library Reference*.

Some example uses of call progress tones are as follows:

- By detecting the ringback tone, the Global Call API can count the rings and report a GCEV_DISCONNECTED event when the call is not answered within the specified number of rings.
- For telephone circuits that include analog links, the local line may not have access to all of the digital signaling information. If so, the user must modify the *.cdp* file accordingly to detect or generate the busy, ringback, or dial tone of the native country.

3.2.3. Call Analysis Functionality for PDK Protocols

PDK protocols configure default call analysis operation through the use of two Protocol Service Layer (PSL) parameters in the protocol *.cdp* file (the parameter names are different for DM3 and Springware boards):

- **PSL_CACallProgressOverride** (parameter for DM3)
  PSL_MakeCall_CallProgress (parameter for Springware): Provides default options for call progress. Possible values are:
  - 0 (Always Off): Specifies that the call progress resource cannot be used by the protocol. This is the default value if this parameter is left undefined in the *.cdp* file.
  - 1 (Preferred): Specifies that the call progress resource is preferred by the protocol. This value is typically used for a T1 and an analog protocol. However, the protocol is able to function without call progress.
  - 2 (Pass-through): Specifies that the call progress resource is configured as specified dynamically by the application, for example, via **gc_MakeCall( )** when using Global Call. This value is typically used by an E1 protocol.
- **PSL_CAMediaDetectOverride** (parameter for DM3)
  PSL_MakeCall_MediaDetect (parameter for Springware): Provides options for media detection. Possible values are:
  - 1 (Preferred): Specifies that the media detection resource is preferred by the protocol. This setting is typically used for a T1 and an analog protocol. The protocol is able to function without media detection.
  - 2 (Pass-through): Specifies that the media detection resource is configured as specified dynamically by the application, for example, via **gc_MakeCall( )** or **gc_SetParm( )** when using Global Call. This value is typically used by an E1 protocol. This is the default value if this parameter is left undefined in the *.cdp* file.

When call progress or media detection support PSL parameters are specified as pass-through values in the *.cdp* file, the application is permitted to define call analysis settings, for example via **gc_MakeCall( )** when using Global Call. More specifically:

- When the **PSL_CACallProgressOverride** (DM3) or **PSL_MakeCall_CallProgress** (Springware) parameter in the *.cdp* file is specified as 2 (Pass-through), the application may disable call progress (the default is enabled) in its call to **gc_MakeCall( ),** for example, when using Global Call.
- When the **PSL_CAMediaDetectOverride** (DM3) or **PSL_MakeCall_MediaDetect** (Springware) parameter in the *.cdp* file is not specified as 1 (Preferred, the default is 2, Pass-through), the application may instead enable media type detection (the default is disabled) in a call to **gc_MakeCall( )** or **gc_SetParm( )** when using Global Call.
- When call progress or media detection support PSL parameters are specified as pass-through values in the *.cdp* file, the application defines call analysis and/or media detection on a per call basis via the **gc_MakeCall( )** or **gc_SetParm( )** call.
- When call analysis behavior is not specified via PSL parameters in the *.cdp* file, the default behavior has call progress always disabled and media type detection disabled by default unless the application explicitly enables media type detection via the **gc_MakeCall( )** or **gc_SetParm( )**.
- If the call progress and/or media type detection parameters are specified in the *.cdp* file as 1 (Preferred) or 0 (Always Off), application setting requests are ignored, for example, the settings specified via **gc_MakeCall( )** or **gc_SetParm( )** when using Global Call.

On Springware boards, PDK protocols support call analysis via the **gc_MakeCall( )** function, which uses the **flags** parameter in the PDK_MAKECALL_BLK structure to determine if call progress and/or media type detection are enabled on a per call basis. The two flags are NO_CALL_PROGRESS and MEDIA_TYPE. The default values are such that call progress is enabled and media type detection is disabled, but the bits in the **flags** parameter can be changed to enable/disable call progress and/or media type detection as required. If this method is used for media detection, the application must receive a GCEV_CONNECTED event before the **gc_GetCallInfo( )**

function can be used to get information about the type of connection. Even after the GCEV_CONNECTED event is received, the call information may not be available. Consequently, the application may need to poll for the information.

On DM3 and Springware boards, PDK protocols also support a more preferred method of call progress configuration using the **gc_SetParm( )** function. The parameters used to specify call progress in this case are **GCPR_MEDIADETECT** (DM3 only) and **GCPR_CALLPROGRESS** (see *Table 2. Global Call Call Progress Settings*). When this method is used to enable media type detection, a GCEV_MEDIADETECTED event is returned to the application on media type detection so that the **gc_GetCallInfo( )** function can be used immediately to get information about the type of connection, that is, the application does not have to wait for a GCEV_CONNECTED event.

When the **gc_GetCallInfo( )** function is used to retrieve information about the detected media type, the **info_id** parameter to the **gc_GetCallInfo( )** function must be CONNECT_TYPE. See the **gc_GetCallInfo( )** function description for a list of the values that may be returned when the **info_id** parameter is CONNECT_TYPE.

Whether a positive media detection result is sufficient to signal a call state change to the CONNECTED state is dependent upon the specific PDK protocol. For example, in PDK protocols where CAS signaling is required for identifying a connection, a signaling bit change must be received before signaling a CONNECTED call state change. For increased flexibility, a separate *.cdp* file parameter, **CDP_Connect_Upon_Media**, may be defined in the associated parameter file and used inside the protocol to enable the protocol to perform a call state change to the Connected state immediately upon positive media detection. This parameter is mostly of interest to T1 protocols.

On Springware boards, call analysis and progress tones are mapped to US specified tones by default. PDK protocols also permit call analysis and progress tones to be customized for non-US defaults via **PSL_TONE_CP_xxx** (where xxx is the call analysis tone type, that is BUSY, RINGBACK, etc.) parameters as specified in the protocol *.cdp* file.

The format of a tone definition in the *.cdp* file is as follows:

```
ALL TONE_t TONE_<NAME> = Frequency_1, Frequency_1_Deviation, Frequency_2,
Frequency_2_Deviation, Amplitude_1, Amplitude_2, OnTime, OnTime_Deviation, OffTime,
OffTime_Deviation, Mode, Repeat Count
```

There are two basic types of tone detection for both single and dual tones: edge detection and cadence detection.

Tone detection using the edge detection algorithm provides notification either when the energy in the specified frequency band(s) exceeds the threshold (leading-edge detection) or no longer exceeds the threshold (trailing-edge detection). Edge detection is identified by assigning a value of zero (0) to the **On Time** parameter. See *Table 4* below.

Tone detection using the cadence detection algorithm provides notification when the energy in the specified frequency band(s) exceeds threshold and meets the timing requirements of the specified ring cadence. Cadence detection, like edge detection, can provide notification either when the cadence completes the specified number of cycles (**Repeat Count** parameter) or when the cadence ceases after ringing the specified number of cycles. Cadence detection is identified by assigning a non-zero value to the **On Time** parameter.

Another tone detection feature is the ability to debounce the leading edge of the tone. Rather than notifying the protocol immediately when the leading edge of the tone is detected, the protocol can specify to wait for a period of time (debounce time) before the tone signal is delivered to the protocol, that is, debouncing. This type of tone detection can be specified in the tone template as:

- **On Time:** plus half the debounce time
- **On Time Deviation:** minus half the debounce time
- **Off Time:** 0
- **Off Time Deviation:** 0
- **Repeat Count:** 0

NOTE: Many Springware boards cannot detect dual tones with frequency components closer than 65 Hz. In these instances, use a single tone template with the specified frequency band (that is, Frequency1 +/- Frequency1 Deviation) encompassing both dual tone ranges.

The meaning of each argument of a tone definition is explained in *Table 4*.

Table 4. TONE_t Signal Definition Parameters

Parameter Number	Name	Description	Detect/ Generate	Edge/ Cadence Detection
1	Frequency 1	Frequency of first tone (in Hz)	Detect, Generate	Edge, Cadence
2	Frequency 1 Deviation	Frequency deviation for first tone (in Hz) **Note:** The minimum recommended value for this parameter is 50.	Detect	Edge, Cadence
3	Frequency 2	Frequency of second tone (in Hz)	Detect, Generate	Edge, Cadence
4	Frequency 2 Deviation	Frequency deviation for second tone (in Hz) **Note:** The minimum recommended value for this parameter is 50.	Detect	Edge, Cadence
5	Amplitude 1	Amplitude of first tone (in dB)	Generate	Neither
6	Amplitude 2	Amplitude of second tone (in dB)	Generate	Neither
7	On Time	On duration (in milliseconds) **Note:** The minimum recommended value is 50.	Detect, Generate	Cadence

Parameter Number	Name	Description	Detect/ Generate	Edge/ Cadence Detection
8	On Time Deviation	On time deviation (in milliseconds) **Note:** The minimum recommended value is 50.	Detect	Cadence
9	Off Time	Off duration (in milliseconds) **Note:** The minimum recommended value is 50.	Detect, Generate	Cadence
10	Off Time Deviation	Off time deviation (in milliseconds) **Note:** The minimum recommended value is 50.	Detect	Cadence

Parameter Number	Name	Description	Detect/ Generate	Edge/ Cadence Detection
11	Mode	Detection notification: • 1 for the onset of the tone. This specifies leading edge in edge detection mode and onset of cadence detection in cadence detection mode. • 0 for the termination of the tone. This specifies trailing edge in edge detection mode and the termination of the cadence after the specified number of cycles in cadence detection mode.	Detect	Edge, Cadence
12	Repeat Count	Repetition count (the number of repetitions on cycles)	Detect, Generate	Cadence

If TONE_x is previously defined, TONE_y may be set equal to TONE_x in the following manner:

```
ALL TONE_t TONE_y = TONE_x
```

The following are examples of tone declarations in a *.cdp* file:

```
/*
This defines the ringback tone. The currently defined tone is a
tone (440Hz+480Hz) on for 0.25 secs and off for 0.25 secs and a
ring count of 1
*/
R4 TONE_t TONE_RINGBACK = 440,0,480,0,0,0,250,0,250,0,0,1

/*
This identifies the KP tone for ANI.
*/
R4 TONE_t TONE_ANI_KP = 1100,0,1700,0,0,0,100,0,0,0,0,1
```

3.2.4. Call Analysis Functionality for ICAPI Protocols

NOTE: The information in this section is applicable to Springware boards only. DM3 boards do not use ICAPI protocols.

Global Call call analysis uses GTD and timers. Some of the country dependent parameters (*.cdp*) files define tone templates for recognition of call progress tones. The tone IDs defined match the protocol parameter numbers (for example, parameter $103 creates tone ID # 103). See the *Voice API Programming Guide* for information about working with and building tone templates.

Parameter $1, $6, or $13 in the *.cdp* file defines the maximum time (in seconds) for a call to be answered. Within that interval, a busy tone and ringback tone can be detected. If the timer expires, the GCEV_DISCONNECTED event is reported to the application.

Two separate busy tones can be defined to accommodate two different call progress failure tones (that is, busy and out-of-order). Busy tones are defined in parameters $103 and $104 using the following format:

```
$103: - <frequency 1> <deviation> <frequency 2> <deviation>
%01: - <on time> <on deviation> <off time> <off deviation>
%02: - <number of cycles before detect>
```

Frequency is expressed in Hz; time duration is expressed in 10 ms units; unspecified values are set to 0. The deviation value for frequency 1 or 2 specifies the allowable variation in Hz. The %01 parameter relates to cadence detection. Cadence detection analyzes the audio signal on the line to detect a repeating

pattern of sound (on time) and silence (off time). The deviation value for cadence detection is the allowable variation in 10 ms units. The %02 parameter specifies the number of times that the cadence on/off pattern must be detected before classifying the tone detected.

To comment out a tone template, insert a ";" (semicolon) as the first character in all three lines of the definition. If either of the busy tones is detected, the GCEV_DISCONNECTED event is reported to the application.

A ringback tone is defined in parameter $105 using the format defined above. The maximum allowable time between successive rings is defined in parameter $3 in 10 ms units. ICAPI starts a timer after receiving a ringback TONEOFF event. Typically, Connect is indicated by line signaling. However, if the network cannot indicate a Connect via line signaling, then Connect can be indicated if the next TONEON event does not arrive before the ICAPI timer expires.

To disable Connect detection, set parameter $3 to 0. Global Call will still be able to count the rings and report the GCEV_DISCONNECTED event if the maximum number of rings is reached. The maximum number of rings is set in parameter $1.

The ringback tone heard on any specific call depends on the specific CO that is serving the called party, not the local CO. If the ringback tone is not known, the recommendation is to remove this tone from the country dependent parameters (*.cdp*) file.

Only the call progress tone definitions in the *.cdp* file are used by the Global Call API. The R1 and R2 tone definitions are used only if you disable R2 MF support in the *icapi.cfg* file by setting the $17 parameter to 1.

The following are examples of the definitions of busy tones $103 and $104 and ringback tone $105 in the *.cdp* file:

```
*****************************
*    TID # 103  BUSY      *
*****************************
$103 BUSY       : 450       35
%01 cadence     : 50 10  50 10
%02 cycle       : 2
```

```
*****************************
*    TID # 104  SBUSY     *
*****************************
$104 SBUSY      :450        35
%01 cadence     : 25  5  25  5
%02 cycle       : 3

*****************************
*    TND # 105  RINGBACK  *
*****************************
$105 RINGBACK   : 450     35
%01 cadence     :  80
```

See the *Voice API Programming Guide* for information about using cadence, cadence detection, and tone definitions for determining the progress of outbound calls.

In addition, the following outbound parameters in the *.cdp* file may need to be modified when using these call progress tones:

- Number of ringback tones before returning GCEV_CALLSTATUS event with a GCRV_NOANSWER result value (typically, parameter $1 or $5)
- Default maximum time in seconds for a call to be answered (typically, parameter $1, $6, or $13)

After the *.cdp* file is modified as described above, whenever one of the defined conditions is detected on a channel, the **gc_MakeCall( )** function is terminated with a busy, no answer, or time-out result/error value.

For some ICAPI protocols, certain parameters must be set in the *.cdp* file to ensure proper operation of the protocol. Refer to the *Global Call Country Dependent Parameters (CDP) Configuration Guide* for country dependent parameters that are most likely to be modified and for any required settings.

NOTE: For ICAPI protocols, the filename specified after @0 in the *.cdp* file must also be specified in the *country.c* file used in Linux applications.

3.3. Header Files

NOTE: The information in this section is applicable to Springware boards only. No additional header files are required for DM3 boards.

In addition to the common Global Call header files *gclib.h* and *gcerr.h* that are required irrespective of the technology used, the following header files may also be required when developing applications that use PDKRT and ICAPI protocols:

- *gcpdkrt.h:* required when using PDK error codes, the PDK_MAKECALL_BLK structure for call analysis, or logging via the **gc_Start( )** function
- *icapi.h:* required when using ICAPI error codes and features

3.4. Resource Association

In E1 CAS and T1 robbed bit protocols, a combination of line signaling and audio tones are used to establish a call. The line signaling is controlled by a network time slot device, or resource, and the tones are controlled by a voice channel (voice resource). Voice channel, voice resource, and tone resource are used interchangeably in this manual when discussing Global Call functionality.

Typically, in E1 CAS or T1 robbed bit environments, a Global Call line device consists of a network time slot resource and a voice resource. When the same voice resource is always used for a given network time slot, then this configuration is called a dedicated voice resource. The Global Call line device ID is a single ID that represents the combination of the voice and network resources that work together to establish and to tear-down calls.

In configurations with more network time slot resources than available voice (or tone) resources, the application may share these available voice resources among the time slots (resource sharing). When voice resources are shared, the Global Call line device ID represents a network time slot after issuing a **gc_OpenEx( )** function. However, before issuing a **gc_MakeCall( )** or a **gc_WaitCall( )** function, a voice resource must be attached to the Global Call line device using the **gc_Attach( )** function and then routed to the line device's network time slot. The **gc_Attach( )** function tells the Global Call protocol handler which voice channel will be used to establish the call. Once the call is established (answered), the application can use this voice resource for other calls by first detaching the voice resource using the **gc_Detach( )** function from the current line device and then attaching this voice resource to another line device using the **gc_Attach( )** function. The **gc_Detach( )** function must not be used to detach the voice resource

until the call is in the connected state. See *Section 5.1. Dedicated Voice Resources* and *Section 5.2. Shared Voice Resources* for more information.

3.5. Alarm Handling

Alarm handling using Global Call is different depending on the board architecture (DM3 or Springware). The following sections provide information about handling alarms in each architecture:

- *Section 3.5.1. Alarm Handling for DM3 boards*
- *Section 3.5.2. Alarm Handling for Springware Boards*

3.5.1. Alarm Handling for DM3 boards

When using DM3 boards, alarms are recognized on a span basis. Once an alarm is detected, all open channels on that span receive a GCEV_BLOCKED event. When the alarm is cleared, open channels receive a GCEV_UNBLOCKED event.

The **gc_SetEvtMsk( )** function can be used to mask events on a line device. Using the **gc_SetEvtMsk( )** function on a line device for a time slot sets the mask for the specified time slot only and does not apply to all time slots on the same trunk as is the case when using Springware boards.

The set of Global Call functions that comprise the Global Call Alarm Management System (GCAMS) interface are supported with the following restrictions:

- Using GCAMS, the application has the ability to set which alarms are blocking and non-blocking as described in the *Global Call API Programming Guide*. However, this capability applies on a span basis only. Changing which alarms are blocking and non-blocking for one time slot results in changing which alarms are blocking and non-blocking for all time slots on the span.
- For T1 technology, the following alarms can be transmitted:
  - YELLOW
  - BLUE

- For E1 technology, the following alarms can be transmitted:
  - Remote alarm - DEA_REMOTE
  - Unframed all 1's alarm - DEA_UNFRAMED1
  - Signaling all 1's alarm - DEA_SIGNALALL1
  - Distant multi-framed alarm - DEA_DISTANTMF
- Using the **gc_GetAlarmParm( )** and **gc_SetAlarmParm( )** functions to retrieve and set specific alarm parameters, for example alarm triggers, is not supported.

The following list shows the alarms that are supported on E1 for DM3 boards. The dagger symbol (†) next to an alarm name indicates that the alarm is blocking by default. The default can be changed using **gc_SetAlarmConfiguration( )**.

- DTE1_CRC_CFA† - Time slot 16 CRC failure
- DTE1_CRC_CFAOK - Time slot 16 CRC failure recovered
- DTE1_FSERR - Received frame sync error
- DTE1_FSERROK - Received frame sync error recovered
- DTE1_LOOPBACK_CFA - Diagnostic mode on the line trunk
- DTE1_LOOPBACK_CFAOK - Diagnostic mode on the line trunk recovered
- DTE1_LOS - Received loss of signal
- DTE1_LOSOK - Received loss of signal recovered
- DTE1_MFSERR - Received multi-frame sync error
- DTE1_MFSERROK - Received multi-frame sync error recovered
- DTE1_RDMA - Received distant multi-frame alarm
- DTE1_RDMAOK - Received distant multi-frame alarm recovered
- DTE1_RED† - Received red alarm
- DTE1_REDOK - Received red alarm recovered
- DTE1_RLOS - Received loss of sync
- DTE1_RLOSOK - Received loss of sync recovered
- DTE1_RRA† - Received remote alarm
- DTE1_RRAOK - Received remote alarm recovered
- DTE1_RSA1 - Received signaling all 1's
- DTE1_RSA1OK - Received signaling all 1's recovered
- DTE1_RUA1 - Received unframed all 1's
- DTE1_RUA1OK - Received unframed all 1's recovered

The following list shows the alarms that are supported on T1 for DM3 boards. The dagger symbol (†) next to an alarm name indicates that the alarm is blocking by default. The default can be changed using **gc_SetAlarmConfiguration( )**.

- DTT1_LOOPBACK_CFA - Diagnostic mode on the line trunk
- DTT1_LOOPBACK_CFAOK - Diagnostic mode on the line trunk recovered
- DTT1_LOS - Initial loss of signal detected
- DTT1_LOSOK - Signal restored
- DTT1_RBL - Received blue alarm
- DTT1_RBLOK - Received blue alarm restored
- DTT1_RCL - Received carrier loss
- DTT1_RCLOK - Received carrier loss restored
- DTT1_RED† - Received a red alarm condition
- DTT1_REDOK - Red alarm condition recovered
- DTT1_RLOS - Received loss of sync
- DTT1_RLOSOK - Received loss of sync restored
- DTT1_RYEL† - Received yellow alarm
- DTT1_RYELOK - Received yellow alarm restored

3.5.2. Alarm Handling for Springware Boards

As described in the *Global Call API Library Reference*, the GCEV_BLOCKED event indicates that a line is blocked and the application cannot issue call-related function calls, and the GCEV_UNBLOCKED event indicates that the line has become unblocked. For example, an alarm condition has occurred or has been cleared, respectively. These events are generated on every opened line device associated with the trunk on which the alarm occurs, if the event is enabled. These events are enabled by default. The application may disable and enable the events by using the **gc_SetEvtMsk( )** function.

Setting the event mask on any line device that represents a time slot will result in setting the mask to the same value on all time slot level line devices on the same trunk. Additionally, setting the event mask on a line device that represents the board will have the same effect (that is, it will set the mask for all time slot level line devices on that trunk).

When an alarm occurs on a Global Call line device, the application must call the **dx_stopch( )** function to stop any application initiated voice processing, such as **dx_play( )** and **dx_record( )**, that is associated with that line device. The

application should wait for the receipt of the GCEV_UNBLOCKED event that signals the end of the alarm condition; then the application can proceed with its call processing (for example, making or receiving calls).

Alarm notification can be configured for time slot devices using the Global Call Alarm Management System (GCAMS). The set of Global Call functions that comprise the GCAMS interface for alarm management is supported. See the *Global Call API Programming Guide* for more information on GCAMS and the *Global Call API Library Reference* for more information on the GCAMS functions.

The following list shows the alarms that are supported on E1 for Springware boards. The dagger symbol (†) next to an alarm name indicates that the alarm is blocking by default. The default can be changed using **gc_SetAlarmConfiguration( )**.

- DTE1_BPVS† - Bipolar violation count saturation
- DTE1_BPVSOK - Bipolar violation count saturation recovered
- DTE1_CECS† - CRC4 error count saturation
- DTE1_CECSOK - CRC4 error count saturation recovered
- DTE1_DPM† - Driver performance monitor failure
- DTE1_DPMOK - Driver performance monitor failure recovered
- DTE1_ECS† - Error count saturation
- DTE1_ECSOK - Error count saturation recovered
- DTE1_FSERR† - Received frame sync error
- DTE1_FSERROK - Received frame sync error recovered
- DTE1_LOS† - Received loss of signal
- DTE1_LOSOK - Received loss of signal recovered
- DTE1_MFSERR† - Received multi-frame sync error
- DTE1_MFSERROK - Received multi-frame sync error recovered
- DTE1_RDMA† - Received distant multi-frame alarm
- DTE1_RDMAOK - Received distant multi-frame alarm recovered
- DTE1_RED - Received red alarm
- DTE1_REDOK - Received red alarm recovered
- DTE1_RLOS† - Received loss of sync
- DTE1_RLOSOK - Received loss of sync recovered
- DTE1_RRA† - Received remote alarm
- DTE1_RRAOK - Received remote alarm recovered
- DTE1_RSA1† - Received signaling all 1’s

- DTE1_RSA1OK - Received signaling all 1's recovered
- DTE1_RUA1† - Received unframed all 1's
- DTE1_RUA1OK - Received unframed all 1's recovered

The following list shows the alarms that are supported on T1 for Springware boards. The dagger symbol (†) next to an alarm name indicates that the alarm is blocking by default. The default can be changed using **gc_SetAlarmConfiguration( )**.

- DTT1_B8ZSD† - Bipolar eight zero substitution detected
- DTT1_B8ZSD - Bipolar eight zero substitution detected recovered
- DTT1_BPVS† - Bipolar violation count saturation
- DTT1_BPVSOK - BPVS restored
- DTT1_DPM† - Driver performance monitor
- DTT1_DPMOK - Driver performance monitor restored
- DTT1_ECS† - Error count saturation
- DTT1_ECSOK - Error count saturation recovered
- DTT1_FERR† - Frame bit error
- DTT1_FERROK - Frame bit error restored
- DTT1_LOS† - Initial loss of signal detected
- DTT1_LOSOK - Signal restored
- DTT1_OOF† - Out of frame error; count saturation
- DTT1_OOFOK - Out of frame restored
- DTT1_RBL† - Received blue alarm
- DTT1_RBLOK - Received blue alarm recovered
- DTT1_RCL† - Received carrier loss
- DTT1_RCLOK - Received carrier loss restored
- DTT1_RED† - Received a red alarm condition
- DTT1_REDOK - Red alarm condition recovered
- DTT1_RLOS† - Received loss of sync
- DTT1_RLOSOK - Received loss of sync restored
- DTT1_RYEL† - Received yellow alarm
- DTT1_RYELOK - Received yellow alarm restored

3.6. Run Time Configuration of the PDKRT Call Control Library

NOTE: The information in this section is applicable to Springware boards only.

Table 5 shows the parameters of the PDKRT call control library that can be configured using the real time configuration management (RTCM) functions. The **gc_GetConfigData( )** function can be used to retrieve the target object configuration, and the **gc_SetConfigData( )** function can be used to update the target object configuration.

NOTE: Since these parameters are statically defined, the **gc_QueryConfigData( )** is not applicable.

Table 5. Configurable PDKRT Call Control Library Parameters

Set ID	Parm ID	Target Object Type	Description	Data Type	Access Attribute*
GCSET_CALLINFO	CALLINFOTYPE	GCTGT_CCLIB_CRN	Calling info type (alternative to **gc_GetCallInfo( )**)	string	GC_R_O
	CATEGORY_DIGIT	GCTGT_CCLIB_CRN	Category digit type (alternative to **gc_GetCallInfo( )**)	char	GC_R_O
	CONNECT_TYPE	GCTGT_CCLIB_CRN	Connect type (alternative to **gc_GetCallInfo( )**)	char	GC_R_O
GCSET_PARM	GCPR_CALLINGPARTY	GCTGT_CCLIB_CHAN	Calling party (alternative to **gc_GetParm( )**/ **gc_SetParm( )**)	string	GC_W_I
	GCPR_LOADTONES	GCTGT_CCLIB_CHAN	Load tones (alternative to **gc_GetParm( )**/ **gc_SetParm( )**)	short	GC_W_I
GCSET_ORIG_ADDR	GCPARM_ADDR_DATA	GCTGT_CCLIB_CHAN	Calling number (alternative to **gc_SetCallNum( )**)	string	GC_W_I

*Access attributes are:
GC_W_I: update
GC_R_O: retrieve only
GC_W_N: update only at null state
GC_W_X: not available

3.7. Run Time Configuration of PDK Protocol Parameters

NOTE: The information in this section is applicable to Springware boards only.

Configurable PDK protocol parameters are grouped into two sets:

- Protocol state information (PSI) variable parameters
- Protocol service layer (PSL) variable parameters

NOTE: To avoid errors, the PSI and PSL parameters of a GCTGT_PROTOCOL_CHAN channel are allowed to be changed only when the channel object does not have an active call.

PSI variable parameters are interpreted by the PDK run-time component (PDKRT). The names of the PSI variable parameters (beginning with CDP_) are found in the *.cdp* file. The PSI parameters that can be accessed via **gc_GetConfigData( )**, **gc_SetConfigData( )**, and **gc_QueryConfigData( )** are protocol dependent. Refer to the *Global Call Country Dependent Parameters (CDP) Configuration Guide* for further information.

The PSL variable parameters are not available to the protocol state machine, but rather are used by the protocol services layer to control the behavior of various network and voice functions. The names of the PSL variable parameters begin with PSL_ and SYS_. No variation in the names is allowed. These parameters are required to control protocol parameters (e.g., timing) or they may control the behavior of the underlying implementation. In the latter case, the parameters will most likely have a platform tag. All of these parameter names must begin with PSL. The PSL parameters that can be accessed via **gc_GetConfigData( )**, **gc_SetConfigData( )**, and **gc_QueryConfigData( )** are shown in *Table 6*.

Table 6. PSL and SYS Parameters

PSL Variable Name	Data Type
PSL_AcceptCallDefaultNumOfRings	Integer
PSL_AnswerCallDefaultNumOfRings	Integer
PSL_MakeCall_CallProgress	Integer
PSL_MakeCall_MediaDetect	Integer

PSL Variable Name	Data Type
PSL_DefaultMakeCallTimeout	Integer
PSL_DXCAS_HOOKFLASH_DURATION	Integer
SYS_FEATURES	String
SYS_PSINAME	String

Table 7 shows the Set ID and Parm ID for these parameter types.

Table 7. Configurable PDK Protocol Parameters

Set ID	Parm ID	Target Object Type	Explanation	Access Attribute**
PDKSET_PSI_VAR *	Dynamically assigned	GCTGT_PROTOCOL_SYSTEM, GCTGT_PROTOCOL_CHAN	Protocol state information (PSI) variable parameters	GC_W_N
PDKSET_SERVICE_VAR	Dynamically assigned	GCTGT_PROTOCOL_SYSTEM, GCTGT_PROTOCOL_CHAN	Protocol service layer (PSL) variable parameter and system parameters	GC_W_N

*Indicates that CAS pattern signals and tones cannot be accessed.
**Access attributes are:
GC_W_I: update
GC_R_O: retrieve only
GC_W_N: update only at null state
GC_W_X: not available

The PDK GCTGT_PROTOCOL_SYSTEM target object is not available until the first **gc_OpenEx( )** function is called to run this protocol.

The Global Call application can call **gc_GetConfigData( )** to retrieve protocol configuration information or **gc_SetConfigData( )** to set protocol configuration information. Since these parameters are protocol dependent, their parameters are dynamically assigned when a protocol is loaded into the PDKRT. Therefore, a Global Call application must call **gc_QueryConfigData( )** to find the parameter

information (set ID, parm ID, and value data type, etc.) first. For more information about these functions, refer to the *Global Call API Programming Guide*.

The pair (target object type, target object ID) supporting **gc_QueryConfigData( )** to find PDKRT protocol parameter information can be one of the following:

- (GCTGT_PROTOCOL_SYSTEM, Global Call protocol ID)
- (GCTGT_PROTOCOL_CHAN, Global Call line device ID)

For a given protocol, although the GCTGT_PROTOCOL_SYSTEM target object and GCTGT_PROTOCOL_CHAN target object share the same set ID and parm ID for PSI variables, they can have different values. When a new GCTGT_PROTOCOL_CHAN target object is opened, it gets a copy of the current PSI variable configuration of GCTGT_PROTOCOL_SYSTEM target object. Under this situation, changes to the GCTGT_PROTOCOL_SYSTEM target object configuration will not affect the configuration of the GCTGT_PROTOCOL_CHAN target object. But the GCTGT_PROTOCOL_SYSTEM target object shares the same PSL variable configuration with other GCTGT_PROTOCOL_CHAN target objects.

The following example shows how to set the **CDP_ANI_ENABLED** parameter for channel ldev running a PDK protocol at the NULL state in asynchronous mode.

NOTE: Error handling is not shown.

```
    GC_PARM t_SourceParm, t_DestParm;
    GC_PARM_ID t_ParmIDSt;
    char t_name[20] = "CDP_ANI_ENABLED";
    long request_id;
    LINEDEV ldev;
    GC_PARM_BLK * t_pParmBlk = NULL;

/* first find the parameter info by calling gc_QueryConfigData() function */
    t_SourceParm.padress = t_name;             /* Pass the PSI variable name */
    memset(&t_ParmIDSt, 0, sizeof(GC_PARM_ID));
    t_DestParm.pstruct = &t_ParmIDStruct;      /* Pass desired the parm info */
    gc_QueryConfigData(GCTGT_PROTOCOL_CHAN, ldev, &t_SourceParm,
                       GCQUERY_PARM_NAME_TO_ID, &t_DestParm);

/* Call GC utility function to insert a parameter data to GC_PARM_BLK */
    gc_util_insert_parm_val(&t_pParmBlk, t_ParmIDStruct.set_ID,
                            t_ParmIDStruct.parm_ID, sizeof(int), 10);
```

```
    /* Call gc_SetConfigData() function to set the “CDP_ANI_ENABLE” */
    gc_SetConfigData(GCTGT_PROTOCOL_CHAN, ldev, t_pParmBlk, 0,
                     GCUPDATE_ATNULL, &request_id, EV_ASYNC);
    ...
    /* Call GC utility function to release the memory after using the GC_PARM_BLK */
    gc_util_delete_parm_blk(t_pParmBlk);
```

3.8. Determining the Protocol Version

NOTE: The information in this section is applicable to Springware boards only.

The following software code demonstrates how you can determine the Global Call protocol version you are running.

```
#include <gclib.h>
#include <gcerr.h>
#include <srllib.h>
int main()
{
     LINEDEV   ldev;
     GC_PARM   parm;
     int retcode;
     METAEVENT metaevent;
     parm.paddress = NULL;
     int mode;
ifdef _WIN32
    mode = SR_STASYNC|SR_POLLMODE;
#else
    mode = SR__POLLMODE;
#endif
     if (sr_setparm(SRL_DEVICE, SR_MODELTYPE, &mode) == -1)
      {
        // Error processing
      }
     gc_Start(NULL);
     retcode = gc_Open(&ldev,":P_na_an_io:N_dtiB1T1:V_dxxxB1C1", 0);
     if (retcode != GC_SUCCESS)
     {
         // Error processing
     }
     sr_waitevt(50);
     retcode = gc_GetMetaEvent(&metaevent);
     if (retcode != GC_SUCCESS)
     {
        // Error processing
     }
     if (metaevent.flags & GCME_GC_EVENT)
     {
         if (metaevent.evttype == GCEV_UNBLOCKED)
     {
              if (gc_GetParm(ldev, GCPR_PROTVER, &parm) ==
                    GC_SUCCESS)
               {
                   printf("The protocol version: %s\n", parm.paddress);
               }
               else
               {
```

```
                    // Error processing
                }
            }
        }

        gc_Close(ldev);
        gc_Stop();
        return(0);
}
```

4. Applying Global Call Functions to E1 CAS or T1 Robbed Bit Applications

Certain Global Call functions have additional functionality or perform differently when used in an E1 CAS or a T1 robbed bit environment. The general function descriptions in the *Global Call API Library Reference* do not contain detailed information on a particular technology. Detailed information in terms of the additional functionality or the difference in performance of those functions in an E1 CAS or a T1 robbed bit environment is contained in this chapter. Note that this information must be used in conjunction with the information presented in the *Global Call API Library Reference*.

The following sections describe how the functions are used specifically in E1 CAS or T1 robbed bit applications. The functions are presented alphabetically. See the *Global Call API Library Reference* for detailed function descriptions.

4.1. gc_AcceptCall( )

On DM3 boards, the **rings** parameter is ignored.

On Springware boards, the **gc_AcceptCall( )** function uses the **rings** parameter to specify the number of rings to wait before terminating the function, that is, before the Global Call API sends the GCEV_ACCEPT event to the application.

- For PDK protocols, if the **rings** parameter is set to 0, the value of the **PSL_AcceptCallDefaultNumOfRings** parameter in the country dependent parameters (*.cdp*) file is used.
- For ICAPI protocols, if the **rings** parameter is set to 0, the value specified in parameter $9 of the country dependent parameters (*.cdp*) file is used.

The **gc_AcceptCall( )** function optionally responds to an inbound call request by providing an indication to the remote end that a call was received but not yet answered. This function causes ringback to be generated.

4.2. gc_AnswerCall( )

On DM3 boards, the **rings** parameter is ignored.

On Springware boards, the **gc_AnswerCall( )** function indicates to the remote end that the connection is established (call has been answered). The **rings** parameter specifies the number of rings to wait before terminating the **gc_AnswerCall( )** function; that is, before answering the call.

- For PDK protocols, if the **rings** parameter is set to 0, the value of the **PSL_AnswerCallDefaultNumOfRings** parameter in the country dependent parameters (*.cdp*) file is used.
- For ICAPI protocols, if the **rings** parameter is set to 0, the value specified in parameter $9 of the country dependent parameters (*.cdp*) file is used.

4.3. gc_BlindTransfer( )

The **gc_BlindTransfer( )** function is only valid for applications using PDK protocols that support call hold and transfer. Check the **sys_features** parameter in the *.cdp* file for a value of Feature_Transfer. See the *Global Call Country Dependent Parameters (CDP) Configuration Guide* for more information.

4.4. gc_CallAck( )

On DM3 boards, the **gc_CallAck( )** function is not supported for E1/T1.

On Springware boards, the **gc_CallAck( )** function may be called before issuing a **gc_AcceptCall( )** or a **gc_AnswerCall( )** function to indicate to the network if more information is desired before completing the call. This function is used to request additional DDI digits from the network. After using this function, call the **gc_GetCallInfo( )** function to retrieve the digits. The **gc_GetCallInfo( )** function will return all DDI digits collected from the network (including both the digits already received and those returned by the network in response to the **gc_CallAck( )** function call). Using the **gc_CallAck( )** function for this service is described in the *Global Call API Library Reference*.

The valid range of values for the **gc_CallAck( )** function **info_len** field is from 1 to GCDG_MAXDIGIT. If more than GCDG_MAXDIGIT digits are required, or if an unknown number of digits is to be requested, set the **info_len** field to GCDG_NDIGIT.

The value GCDG_PARTIAL may be ORed with the number of digits field if the application needs to call the **gc_CallAck( )** function again for this call (that is, if the application needs additional DDI digits before accepting or rejecting the call).

4.5. gc_Close( )

The **gc_Close( )** function only affects the link between the calling process and the device. For CAS protocols, if a voice resource is currently assigned to the specified line device, the voice resource will be closed. To keep the voice resource open for other operations, use the **gc_Detach( )** function to detach the voice resource from the line device before issuing the **gc_Close( )** function.

Functionality of **gc_Close( )** is different for Springware and DM3 boards with regards to stopping the protocol. Springware boards stop the protocol after **gc_Close( )**; DM3 boards do not.

On DM3 boards, **gc_Close( )** typically sets the protocol out of service; the protocol is not stopped until the board is stopped. Therefore, when a DM3 board uses a protocol that includes the CDP_ProtocolStopsOffhook parameter, which determines the state of the hook switch signaling (on-hook or off-hook) when the protocol stops after **gc_Close( )**, this parameter has no effect.

4.6. gc_CompleteTransfer( )

The **gc_CompleteTransfer( )** function is only valid for applications using PDK protocols that support call hold and transfer. Check the **sys_features** parameter in the *.cdp* file for a value of Feature_Transfer. See the *Global Call Country Dependent Parameters (CDP) Configuration Guide* for more information.

4.7. gc_Detach( )

The **gc_Detach( )** function logically disconnects a voice channel from a line device. It is the responsibility of the application to make sure that there is a voice resource available while the **gc_WaitCall( )** function is active and that the current Global Call call state is Null or Idle. Furthermore, the **gc_Detach( )** function can only be called in the Null, Idle, or Connected states.

4.8. gc_DropCall( )

The **gc_DropCall( )** function supports the following values for its **cause** parameter:

- GC_CALL_REJECTED: call is not accepted
- GC_NETWORK_CONGESTION: cannot establish connection due to volume of traffic on network
- GC_NORMAL_CLEARING: normal end of call
- GC_SEND_SIT: sends a special information tone
- GC_UNASSIGNED_NUMBER: invalid called party number
- GC_USER_BUSY: called party is busy

CAUTION

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( )**.

Some protocols do not support all **gc_DropCall( )** causes for dropping a call. Any unsupported cause(s) is automatically mapped to the most appropriate cause. This approach facilitates developing protocol independent applications.

From the Accepted state, some protocols do not support a forced release of the line; that is, issuing a **gc_DropCall( )** function after a **gc_AcceptCall( )** function. Refer to the Protocol Limitations section in the *Global Call Country Dependent*

Parameters (CDP) Configuration Guide for your protocol. If a forced release is attempted, the function fails and an error is returned. To recover, the application should issue a **gc_AnswerCall( )** function followed by **gc_DropCall( )** and **gc_ReleaseCall( )** functions. However, anytime a GCEV_DISCONNECTED event is received in the Accepted state, the **gc_DropCall( )** function can be issued.

After the **gc_AnswerCall( )** function is issued, the application must wait for a GCEV_ANSWER event. Otherwise the **gc_DropCall( )** function is ignored, no error is returned, and no drop call action is taken.

When using ICAPI protocols (Springware only), the **gc_DropCall( )** function occasionally results in the generation of the GCEV_DROPCALL event followed by a GCEV_BLOCKED event. The generation of the GCEV_BLOCKED event is most likely if the **gc_DropCall( )** function is issued before the call is connected. The reason for the GCEV_BLOCKED event is that the remote side does not recognize the disconnection in a timely manner. When the GCEV_BLOCKED event occurs, call-related Global Call functions should not be issued until a GCEV_UNBLOCKED event is detected on the respective device.

In some protocols, a **gc_DropCall( )** command on a call in the Accepted state requires a momentary transition to the Connected state. This may result in a charge being registered for the call.

4.9. gc_Extension( )

4.9.1. gc_Extension( ) with DM3 Boards

On DM3 boards, the **gc_Extension( )** function can be used to access the functionality of the Direct Signaling protocol. The Direct Signaling protocol is not a call control protocol; it is used strictly to give applications direct control over the signaling patterns on a line, as a means to allow the application to implement its own protocols. The Direct Signaling protocol allows the application to generate and detect signaling patterns. Applications can call the **gc_Extension( )** function to generate up to 11 distinct CAS patterns, and through the GCEV_EXTENSION event, be notified when one of the patterns is detected by the protocol. For details about the Direct Signaling protocol and the **gc_Extension( )** function, see the *Global Call Country Dependent Parameters (CDP) Configuration Guide*.

4.9.2. gc_Extension( ) with Springware Boards

On Springware boards, the **gc_Extension( )** function provides the capability to dynamically configure the behavior of a PDK protocol in a manner that is specific to that protocol. The **gc_Extension( )** function is only valid for applications using PDK protocols that support this feature. Check the associated *.cdp* file or the *Global Call Country Dependent Parameters (CDP) Configuration Guide* for more information regarding the support and configuration of this feature including its usage.

For PDK protocols that support this feature, up to a maximum of three integer values and two strings may be passed to the protocol by configuring the **parmp** parameter as a pointer to a structure of type PDK_EXT_SIG_BLK:

```
struct pdk_ext_sig_blk {
     int  int1Val;
     int  int2Val;
     int  int3Val;
     char *str1Val;
     char *str2Val;
} PDK_EXT_SIG_BLK;
```

The actual meaning and usage of these integers and string values are protocol specific. The PDKRT library only supports this function in asynchronous mode (EV_ASYNC), and therefore the **retvalp** parameter is ignored.

If the protocol requires unsolicited notification of the application as a result of the completion of a feature triggered by the **gc_Extension( )** function, or for any other asynchronous notification for any documented reason, it may send an extension event.

The extension event, GCEV_EXTENSION, is provided to support unsolicited events resulting from PDK protocol-specific features. Refer to the *Global Call Country Dependent Parameters (CDP) Configuration Guide* for the proper handling of any unsolicited extension events as well as the format of the event, that is, identifier mappings and contents of the data buffer.

For this event, the pointer to the extended event data block, exteventdatap, within the METAEVENT structure, points to a structure, EXTENSIONEVTBLK, defined as follows:

```
typedef struct{
     unsigned char ext_id;      /* unused for PDK protocols */
     GC_PARM  gcparm;           /* GC_PARM union to provide technology-specific info */
     short int parm;
} EXTENSIONEVTBLK;
```

The **gcparm** field of the METAEVENT structure itself points to a structure of type PDK_EXT_SIG_BLK (above) so that three integers and two string values may be returned from the protocol to the application to indicate protocol-specific status information.

The extension block structure, EXTENSIONEVTBLK, referenced in a GCEV_EXTENSION event, has a persistence only until the next call of the **gc_GetMetaEvent( )** function. In other words, any information contained or referenced in a GCEV_EXTENSION event must be either processed or cached within the application, or risk being lost upon the next call of the **gc_GetMetaEvent( )** function.

4.10. gc_GetCallInfo( )

For E1 CAS and T1 robbed bit protocols that support enhanced call analysis (call progress), the **gc_GetCallInfo( ) info_id** CONNECT_TYPE parameter contains the type of connection as returned by the function. These connection types are:

- GCCT_CAD: connection due to cadence break
- GCCT_PVD: connection due to voice detection
- GCCT_PAMD: connection due to answering machine detection
- GCCT_FAX:- connection due to fax machine detection
- GCCT_NA: connection type is not available

PDK protocols provide support for enhanced call analysis.

For protocols that do not support enhanced call progress analysis, the **gc_GetCallInfo( )** function with the CONNECT_TYPE parameter specified will return a CONNECT_TYPE value of GCCT_NA (not available).

For E1 CAS protocols that support the CALLINFOTYPE **info_id** parameter, a call information string containing either a CHARGE or NO CHARGE value is returned by the parameter; check your protocol in the *Global Call Country Dependent Parameters (CDP) Configuration Guide* for applicability.

On Springware boards, the **gc_GetCallInfo( )** function can be used to retrieve the CATEGORY_DIGIT parameter for E1 CAS calls. (CATEGORY_DIGIT is not supported on DM3 boards.)

4.11. gc_GetParm( )

The **gc_GetParm( )** function retrieves the value of the specified parameter for a line device. In addition to the GCPR_CALLINGPARTY parameter, which is common across all technologies and documented in the *Global Call API Library Reference*, the following parameters are supported:

- On DM3 boards:
  - GCPR_CALLPROGRESS
  - GCPR_MEDIADETECT
  - GCPR_MINDIGITS
- On Springware boards:
  - GCPR_CALLPROGRESS
  - GCPR_LOADTONES

See *Section 4.20. gc_SetParm( )* for more information on the meaning of these parameters.

4.12. gc_HoldCall( )

The **gc_HoldCall( )** function is only valid for applications using PDK protocols that support call hold and transfer. Check the **sys_features** parameter in the *.cdp*

file for a value of Feature_Hold. See the *Global Call Country Dependent Parameters (CDP) Configuration Guide* for more information.

4.13. gc_MakeCall( )

4.13.1. Use of the timeout Parameter

When using E1 CAS or T1 robbed bit line devices, the **timeout** parameter in the **gc_MakeCall( )** function is supported when the **mode** parameter is set to either EV_SYNC or EV_ASYNC.

For ICAPI protocols (Springware only), when the **mode** parameter is set to EV_ASYNC, the **timeout** parameter overrides the time-out parameter ($13) value and the outbound number of ringback tones parameter ($1) in most protocol country dependent parameters (*.cdp*) files.

- If the **timeout** parameter is set to 0, then the time-out and ringback parameters in the *.cdp* file are used to set the time-out conditions.
- If the **timeout** parameter is set to a value larger than a protocol time-out value, a protocol time-out may occur first, which will cause the **gc_MakeCall( )** function to fail. The protocol time-out is configured in the country dependent parameters (*.cdp*) file.
- If the **timeout** value is reached before the remote end answers the call, the application is notified of this condition and should respond as described in the **gc_MakeCall( )** function description in the *Global Call API Library Reference*.
- If all **timeout** values are set to 0, no time-out condition will apply.

For PDK protocols, the time-out value used is determined by:

- The **timeout** parameter in the **gc_MakeCall( )** function.
- The **PSL_DefaultMakeCallTimeout** parameter specified in the *.cdp* file if the **timeout** parameter in the **gc_MakeCall( )** function is 0 and call analysis is not specified.

- The **PSL_CallProgressMaxDialingTime** parameter specified in the *.cdp* file if the **timeout** parameter in the **gc_MakeCall( )** function is 0 and call analysis is specified.

NOTE: PDK protocols do not use the outbound number of ringback tones to define the time-out.

4.13.2. Other gc_MakeCall( ) Considerations

If your T1 robbed bit circuit is provisioned for Feature Group A, your application should call the **gc_MakeCall( )** function with a null dial string.

When using R2 protocols, a "#" in the dial string is not supported.

If a protocol error occurs during dialing and the default call progress enabled governs, then an error code or an event is returned as described in the **gc_MakeCall( )** function description in the *Global Call API Library Reference*. If call progress is disabled and a protocol error occurs during dialing, then a GCRV_BUSY result value or an EGC_BUSY error is returned.

For drop and insert applications, call progress is typically disabled to enable the application to complete the dialing sequence, listen for voice or ringback on the line, and:

- if ringback is detected, to transition to the Alerting state
- if voice is detected, to transition to the Connected (call answered) state and to implement voice cut-through immediately

This methodology enables the application to pass signaling from the remote end (outbound line) to the caller on the inbound line. If call progress is not disabled, then the GCEV_ALERTING event and the GCEV_ANSWERED event may be received from the outbound line for an unacceptable amount of time after the dialing sequence was completed. During this period of time, the caller could misinterpret the silence on the line as a disconnect or a failure, and then hang up and redial. For further information, see the tips for programming drop and insert applications in the *Global Call API Programming Guide*.

In an E1 environment, the GCEV_ALERTING event is generated when the equivalent of ringback is recognized. For almost all E1 protocols, this is a

required part of the protocol, so E1 applications will receive the GCEV_ALERTING event by default.

In a T1 environment, the GCEV_ALERTING event is generated when the ringback is recognized. However, not all inbound applications will generate a ringback tone; for example, the PDK US MF protocol has disabled ringback tone generation by default to minimize call setup time. (Detecting the ringback tone takes several tenths of a second.) If the outbound application does not wish to use the detection of the ringback tone to generate the GCEV_ALERTING event, the **CDP_OUT_Send_Alerting_After_Dialing** parameter in the *pdk_us_mf_io.cdp* file should be set to 1 (default is 0). That way, if call progress is enabled, GCEV_ALERTING is sent after dialing is initiated rather than when ringback is detected.

Since GCEV_ALERTING is an optional event triggered by the inbound side, all applications must be able to handle not receiving the GCEV_ALERTING event.

When the **gc_MakeCall( )** function sets up a call, the default is to enable call analysis (call progress). To change the enabled call progress default when making a call on Springware boards, see *Section 4.13.3. PDK_MAKECALL_BLK* for PDK protocols and *Section 4.13.4. IC_MAKECALL_BLK* for ICAPI protocols. (These structures do not apply to DM3 boards, which use **gc_SetParm( )** parameters to change call progress configuration as discussed in *Section 3.2.1. Call Analysis with DM3 Boards*.)

4.13.3. PDK_MAKECALL_BLK

For Springware boards using PDK protocols, when the **gc_MakeCall( )** function sets up a call, the default is to enable call analysis (call progress). This default can be changed on a call basis by setting the **flags** parameter in the PDK_MAKECALL_BLK data structure. See *Table 8. PDK_MAKECALL_BLK Field Descriptions*.

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

```
typedef struct pdk_makecall_blk{
   unsigned long         flags;
   void                  *v_rfu_ptr;
   unsigned long         ul_rfu[4];
}PDK_MAKECALL_BLK;
```

Table 8. PDK_MAKECALL_BLK Field Descriptions

Field	Description
flags	Controls call analysis and media type detection on a per call basis. The flags included are: • NO_CALL_PROGRESS: Set to 0 to enable call analysis (default). Set to 1 to disable call analysis. • MEDIA_TYPE_DETECT: Set to 0 to enable media type detection. Set to 1 to disable media type detection.
*v_rfu_ptr	Reserved for future use.
ul_rfu[4]	Reserved for future use.

4.13.4. IC_MAKECALL_BLK

For ICAPI protocols, when the **gc_MakeCall( )** function sets up a call, the default is to enable call analysis (call progress). This default can be changed on a call basis by setting the **flags** parameter in the IC_MAKECALL_BLK data structure. See *Table 9. IC_MAKECALL_BLK Field Descriptions*.

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

```
typedef struct ic_makecall_blk{
   unsigned long         flags;
   void                  *v_rfu_ptr;
   unsigned long         ul_rfu[4];
}IC_MAKECALL_BLK;
```

Table 9. IC_MAKECALL_BLK Field Descriptions

Field	Description
flags	Controls call analysis and media type detection on a per call basis. The flags included are: • NO_CALL_PROGRESS: Set to 0 to enable call analysis (default). Set to 1 to disable call analysis.
*v_rfu_ptr	Reserved for future use.
ul_rfu[4]	Reserved for future use.

4.14. gc_OpenEx( )

The **gc_OpenEx( )** function is used to open both network board and channel (time slot) devices. This generic call control function initializes the specified time slot on the specified trunk. A line device ID will be returned to the application. The E1 or T1 feature of this function specifies the voice device as part of the **devicename** parameter.

4.14.1. Conventions for Specifying the devicename Parameter

A device is specified by the **devicename** parameter using a format that includes protocol specific information.

The format for the fields used to specify this parameter is:

```
:N_<network_device_name>:P_<protocol_name>:V_<voice_channel_name>
```

The prefixes (N_, P_, and V_) are used for parsing purposes. These fields may appear in any order. The fields within the **devicename** parameter must each begin with a colon.

The conventions described below allow the Global Call API to map subsequent calls made on specific line devices or CRNs to interface-specific libraries.

- <**network_device_name**>: This field is required. It may be a board name or a time slot name:
  - If <network_device_name> is a board name, use the format: dtiB<number of board>.
  - If <network_device_name> is a time slot name, use the format: dtiB<number of board>T<number of time slot>.
- <**protocol_name**>: This field is required on Springware boards. It specifies the protocol to use. Use the root file name of the country dependent parameters (*.cdp*) file.

  NOTE: On DM3 boards, the protocol is determined at board initialization time and not when a Global Call device is opened. For compatibility, the <protocol_name> field may be specified, but it is not used.
- <**voice_channel_name**>: This field is optional depending on your application (see *Section 5.1. Dedicated Voice Resources* or *Section 5.2. Shared Voice Resources*). It specifies the name of the voice channel to be associated with the device being opened. Use the following format: dxxxB<virtual board number>C<channel number>.

  NOTE: Attachment to different types of DM3 voice devices is dependent on the protocol downloaded. For example, if one board has ISDN for protocols and another has T1 CAS, the T1 CAS network devices cannot be attached to the voice devices on the ISDN board. See the *Global Call API Programming Guide* for further information.

4.14.2. Examples of the devicename Parameter

The following examples illustrate the **devicename** parameter when using E1 ICAPI protocols:

- For a D/300SC-E1 board specified as dtiB3, the **devicename** for time slot 1 on this board using the inbound Brazil protocol is:

```
:N_dtiB3T1:P_br_r2_i
```

- To open both voice channel 2 on virtual voice board 4 and the above network device, the **devicename** for the inbound Brazil protocol is:

```
:N_dtiB3T1:P_br_r2_i:V_dxxxB4C2
```

- A line device may represent a board, thus to open the board dtiB1, the **devicename** is:

```
:N_dtiB1:P_br_r2_i
```

4.14.3. Other gc_OpenEx( ) Considerations

For E1 CAS or T1 robbed bit applications, always specify a network resource (board or time slot level) and a protocol. A voice resource (<voice_channel_name>) may also be specified for E1 CAS or T1 robbed bit operations. When a voice resource is specified, Global Call automatically opens the voice device and internally attaches the voice device to the line device.

When using the CT Bus and a voice resource is specified, the **gc_OpenEx( )** function routes the voice and network resources together.

When the voice resource is not specified, the application must perform these functions (open device, route, attach); see *Section 5.1. Dedicated Voice Resources* and *Section 5.2. Shared Voice Resources* for details.

When a network resource is specified, the **gc_OpenEx( )** function internally issues a **dt_open( )** function. Likewise, when a voice resource is specified, the **gc_OpenEx( )** function internally issues a **dx_open( )** function. The corresponding network or voice device handle may be retrieved using the **gc_GetResourceH( )** function. These lower level device handles may be useful for routing or for playing or recording a file.

If a **gc_OpenEx( )** function fails with an error value of EGC_DXOPEN, then the internally issued **dx_open( )** function failed. If a **gc_OpenEx( )** function fails with an error value of EGC_DTOPEN, then the internally issued **dt_open( )** function failed.

4.14.4. Handling GCEV_BLOCKED and GCEV_UNBLOCKED Events

At the firmware level, when using Springware boards, the line is considered **unblocked** until otherwise informed (that is, some event occurs to change the state). From the Global Call perspective, the line is considered **blocked** until otherwise informed. To reconcile this difference in behavior, the Global Call software generates the required GCEV_UNBLOCKED event as part of the **gc_OpenEx( )** functionality with Springware boards.

When using Springware boards, if a blocking alarm exists on the line when an application tries to open a device, the **gc_OpenEx( )** function will complete, generating the GCEV_UNBLOCKED event, before the firmware detects that the alarm exists, which would trigger the generation of a GCEV_BLOCKED event. This means that the application temporarily sees a GCEV_UNBLOCKED event even though an alarm exists on the line. The application must be capable of handling a GCEV_BLOCKED event at any time, even milliseconds after a GCEV_UNBLOCKED event.

4.15. gc_ResetLineDev( )

On Springware boards, for applications that use PDK protocols, the **gc_ResetLineDev( )** function cannot be called while there is an alarm on the line.

On Springware boards, for applications that use ICAPI protocols, the **gc_ResetLineDev( )** function is not supported in synchronous mode. If the application calls **gc_ResetLineDev( )** on a line device in synchronous mode (that is, with the mode parameter set to EV_SYNC), the function fails silently.

There are no restrictions on using **gc_ResetLineDev( )** with DM3 boards.

4.16. gc_RetrieveCall( )

The **gc_RetrieveCall( )** function is only valid for applications using PDK protocols that support call hold and transfer. Check the **sys_features** parameter in the *.cdp* file for a value of Feature_Hold. See the *Global Call Country Dependent Parameters (CDP) Configuration Guide* for more information.

4.17. gc_SetBilling( )

On DM3 boards, the **gc_SetBilling( )** function is not supported for E1/T1.

On Springware boards, the **gc_SetBilling( )** function sets different billing rates on a per call basis. For example:

- To charge the call, use **gc_SetBilling(crn, GCR_CHARGE, NULL, EV_SYNC)**
- To select no-charge for the call, use **gc_SetBilling(crn, GCR_NOCHARGE, NULL, EV_SYNC)**

The **gc_SetBilling( )** function is called after the GCEV_OFFERED event arrives and before issuing a **gc_AcceptCall( )** or **gc_AnswerCall( )** function.

Not all protocols support this feature; see the *Global Call Country Dependent Parameters (CDP) Configuration Guide* for protocol specific limitations.

The **mode** parameter must be set to EV_SYNC. Asynchronous mode (EV_ASYNC) is not supported for this function.

4.18. gc_SetChanState( )

The GCLS_INSERVICE and GCLS_OUT_OF_SERVICE states are the only valid service states that can be used to set the state of a line in an E1 CAS or T1 robbed bit environment.

4.19. gc_SetEvtMsk( )

On DM3 and Springware boards using PDK protocols, all of the mask parameter values are supported. See the **gc_SetEvtMsk( )** function reference page in the *Global Call API Library Reference* for more information.

On Springware boards using ICAPI protocols, the following mask parameter values are supported:

- GCMSK_ALERTING

- GCMSK_BLOCKED
- GCMSK_UNBLOCKED

4.20. gc_SetParm( )

The **gc_SetParm( )** function sets the default parameters and all channel information associated with the specific line device. In addition to the GCPR_CALLINGPARTY parameter, which is common across all technologies and documented in the *Global Call API Library Reference*, the parameters listed in *Table 10* are supported.

Table 10. Parameters Supported, gc_GetParm( ) and gc_SetParm( )

Parameter	Level	Description	Supported on
GCPR_CALL PROGRESS	channel	Enables or disables call progress; enabled by default. If this parameter is disabled, post-connect call progress is also disabled, regardless of the setting of GCPR_MEDIADETECT.	DM3, Springware
GCPR_LOA DTONES	channel	Enables or disables downloading of predefined call progress tones to the firmware. These tones are predefined in the E1 CAS or T1 robbed bit specific configuration files and are used for call progress.	Springware
GCPR_MEDI ADETECT	channel	Enables or disables post-connect call progress or media detection; disabled by default.	DM3
GCPR_MIN DIGITS	channel	Specifies the minimum number of digits to receive before a call is offered to the application.	DM3

For further information about the GCPR_LOADTONES parameter, see *Section 3.1. Global Tone Detection (GTD) Tone Considerations*. The tones are downloaded during execution of the **gc_Attach( )** or **gc_AttachResource( )** function.

For further information about the GCPR_CALLPROGRESS and GCPR_MEDIADETECT parameters, see *Section 3.2. Call Progress and Call Analysis*.

4.21. gc_SetUpTransfer( )

The **gc_SetUpTransfer( )** function is only valid for applications using PDK protocols that support call hold and transfer. Check the **sys_features** parameter in the *.cdp* file for a value of Feature_Transfer. See the *Global Call Country Dependent Parameters (CDP) Configuration Guide* for more information.

4.22. gc_Start( ) and gc_Stop( )

On Springware boards using PDK protocols, the **gc_Start( )** function is used to access the error and debug logging capabilities of the PDKRT call control library. See *Section 7.1. Debugging Applications that Use PDK Protocols* for more information.

On Springware boards using ICAPI protocols, when the **gc_Start( )** function is called, a log file, if enabled, is created. This file logs debug information for all ICAPI call control libraries for all open channels. The log file remains open until the **gc_Stop( )** function is called. This allows channels to be opened, closed, and reopened multiple times without overwriting or otherwise affecting the continuity of the log file. See *Section 7.2. Debugging Applications that Use ICAPI Protocols* for more information.

4.23. gc_StartTrace( )

On DM3 boards, the **gc_StartTrace( )** function is not supported for E1/T1.

On Springware boards using PDK protocols, the **gc_StartTrace( )** function can be used to enable logging on individual channels. This function has no effect

unless the name of the log file and the logging level have been set using the **gc_Start( )** function. The **gc_StartTrace( ) filename** parameter is ignored. The name of the log file is specified in the PDK_START_STRUCT data structure. See *Section 7.1. Debugging Applications that Use PDK Protocols* for more information.

4.24. gc_SwapHold( )

The **gc_SwapHold( )** function is only valid for applications using PDK protocols that support call hold and transfer. Check the **sys_features** parameter in the *.cdp* file for a value of Feature_Transfer. See the *Global Call Country Dependent Parameters (CDP) Configuration Guide* for more information.

5. Resource Allocation and Routing

E1 CAS and T1 robbed bit protocols require tone generation and detection capability and therefore require a voice or tone resource for setting up a call. Application development considerations for using dedicated voice (or tone) resources or shared voice (or tone) resources in an E1 CAS and T1 robbed bit environment are discussed in this chapter.

5.1. Dedicated Voice Resources

Applications requiring voice resources during the entire call (for example, voice-mail and announcements) must have enough voice channels to dedicate one channel to each network interface time slot. Global Call simplifies applications written to handle E1 CAS and T1 robbed bit protocols using dedicated voice resource configurations. To use Global Call functionality to set up dedicated resources, the application must pass both the network time slot and the voice channel to the **gc_OpenEx( )** function. The Global Call API uses this information to automatically:

- open the network board device, if not previously opened (the board device is used internally by Global Call)
- open both the voice channel and the network time slot
- route the voice channel and network time slot together (full duplex) (CT Bus configurations only)
- associate the voice channel and the network time slot by issuing an internal **gc_Attach( )** function

For CT Bus applications, applications using dedicated voice resources (a voice resource dedicated to a network resource) do not need to route the voice and network resources together nor issue the **gc_Attach( )** function before making a call or when handling a pending call. For applications using shared voice resources, the voice resource must be attached to a network resource before call establishment. After call establishment, this voice resource may be detached and then attached to a different network resource.

To perform activities such as routing and voice store and forward, etc., use the **gc_GetResourceH( )** function to obtain the voice and network handles associated with a line device. For example, before playing a file, you can retrieve the voice handle using the **gc_GetResourceH( )** function. If needed, you may route other resources to the network interface (for example, to send a fax) and reroute the voice channel back to the network interface before setting up or waiting for another call. You must route the same voice channel back to the associated network interface channel because these two resources were internally attached when opened.

The following example illustrates the function calls that apply when using dedicated voice resources.

5.1.1. Dedicated Voice Resources Example

```
            .
            .
            #define MAXCHAN  30
            struct linebag{
               LINEDEV   ldev;
               CRN       crn;
               INT       state;
            }port[MAXCHAN+1]
            .
            .
            /* Open a Global Call device with a voice channel and a
               network time slot */
1 ---->     if (gc_OpenEx(&linedev, ":N_dtiB1T1:P_br_r2_o:V_dxxxB1C1", 0,
                (void*)&port[port_index]) == GC_SUCCESS) {
                     /*
                      * Wait for GCEV_UNBLOCKED event.
                      */
                     .
                     .
                     /* Make an outgoing call */

2 ---->              if (gc_MakeCall(linedev, &crn, "123456", NULL, 0,
                         EV_ASYNC) == GC_SUCCESS) {
                              /*
                               * Wait for GCEV_CONNECTED event.
                               */
                     } else {
                          /* Process error from gc_MakeCall( ) */
                     }
            } else {
                   /* Process error from gc_Open( ) */
            }
            .
            .
```

Legend:

1 The **gc_OpenEx( )** function:

- opens a Global Call line device using time slot 1 of dtiB1, opens voice channel dxxxB1C1, and configures the line device to use outbound Brazilian R2 protocol
- opens the time slot and voice channel automatically
- opens the network board device automatically, if not already opened to monitor the alarm
- sets the user attribute, **usrattr**, (void*)&port[port_index] into the channel information structure

CT Bus time slot routing and attaching are done automatically. The function need only be called once for a time slot/voice channel pair.

2 The **gc_MakeCall( )** function is invoked once for each outbound call.

5.2. Shared Voice Resources

Applications requiring voice resources for a limited portion of the call, typically during call setup, may share voice resources among the available network time slots. For example, using a D/320SC voice board and two DTI/300SC (E1 interface) network boards, 32 voice channels may be able to handle the audio portions of the call control for 60 network interface time slots. This savings in hardware requires more complexity in writing the application, which must manage both the voice and network resources, and places limits on your call throughput. The number of calls that can be established simultaneously is limited to the number of voice resources in the system.

For voice resource sharing configurations, you need only specify the network time slot and protocol in the **gc_OpenEx( )** function. This function uses the Global Call library to open the network time slot device. Your application must also open the voice device and then route and attach the necessary resources before these resources are needed for signaling. You must explicitly open the voice device by issuing a **dx_open( )** function to open the voice device selected. For routing these devices, use the native time slot routing functions or the CT Bus **nr_scroute( )** and **nr_scunroute( )** functions provided for the voice and network devices. For example, route the devices using the routing functions provided by the network and voice libraries, and then use the **gc_Attach( )** and **gc_Detach( )** functions to associate or disassociate a voice channel and a Global Call line device and therefore a network time slot. When the above sequence of operations completes, use the **gc_MakeCall( )** or **gc_WaitCall( )** function, as appropriate.

After a call is answered, the voice resource can be detached from the network time slot using the **gc_Detach( )** function and routed to another network time slot using the **nr_scunroute( )** and **nr_scroute( )** functions.

The following example illustrates the function calls that apply when using shared voice resources.

5.2.1. Shared Voice Resources Example

```
1 ---->     if (gc_OpenEx(&linedev, ":N_dtiB1T1:P_br_r2_o, 0,
               (void*)&port[port_index]) == GC_SUCCESS) {
                        /* process error */
            }

2 ---->     if (gc_GetNetworkH(linedev, &networkh) != GC_SUCCESS)
            {
                        /* process error */
            }

3 ---->     if ((voiceh = dx_open("dxxxB1C1", 0)) == -1)
            {
                       /* process error */
            }

4 ---->     printf("******* %d: Calling Attach %d\n", index, voiceh);
            if (gc_Attach(linedev, voiceh, EV_SYNC) != GC_SUCCESS)
            {
                        /* process error */
            }

5 ---->     if (nr_scroute(networkh, SC_DTI, voiceh, SC_VOX,
                    SC_FULLDUP) == -1)
                {
                          /* process error */
                }
                  /* Wait for GCEV_UNBLOCKED event */

6 ---->     if (gc_MakeCall(linedev, &crn, "123456", NULL, 0, EV_ASYNC)
                       != GC_SUCCESS)
                {
                         /* process error */
                }
            .
            .
            /*
             * Wait for GCEV_CONNECTED event. Voice resource may be detached
             * if necessary after receiving this event.
             */

7 ---->     if (gc_Detach(linedev, voiceh, EV_SYNC) != GC_SUCCESS)
                {
                        /* process error */
                }

8 ---->     if (nr_scunroute(networkh, SC_DTI, voiceh, SC_VOX, SC_FULLDUP) == -1)
            {
                        /* process error */
            }
```

Legend:

1 The **gc_OpenEx( )** function:

- opens a Global Call line device using time slot 1 of dtiB1 using outbound Brazilian R2 protocol

- opens the network board device automatically, if not already opened
- sets the user attribute, **usrattr**, (void*)&port[port_index] into the channel information structure

The specified network time slot device is opened. This function need only be called once for a time slot.

2 The **gc_GetNetworkH( )** function retrieves network device handle.

3 The **dx_open( )** function opens voice device and gets voice device handle.

4 The **gc_Attach( )** function logically connects voice and network resources.

5 The **nr_scroute( )** function routes voice and network resources together.

6 The **gc_MakeCall( )** function is invoked each time a call is to be made.

7 The **gc_Detach( )** function disassociates the voice resource from the Global Call line device.

8 The **nr_scunroute( )** function unroutes the voice and network resources.

6. Protocols

The protocols supported, protocol naming conventions, protocol components, and their corresponding protocol files are described in this chapter.

NOTES:

1. See the documentation accompanying the Global Call Protocols package for more information about using the protocols and the country dependent parameter (CDP) files. The *Global Call Country Dependent Parameters (CDP) Configuration Guide* includes detailed procedures for configuring country dependent parameters and for downloading the protocol and CDP file.
2. The development of the ICAPI protocols supported by Global Call has been capped. Customers should migrate to equivalent protocols developed using the Protocol Development Kit (PDK). New protocol development as well as existing protocol support will be on the PDK. ICAPI protocols are supported only on Springware boards. PDK protocols are supported on both DM3 boards and Springware boards.

6.1. Protocols Supported

Protocols are distributed separately on individual CDs or as part of a software package release. This modular design simplifies adapting applications for use in numerous countries. The Global Call protocols available are listed in the *Global Call Country Dependent Parameters (CDP) Configuration Guide*. For the most up-to-date list of available protocols, contact your nearest Intel Sales Office.

The protocol and parameters used at the application's interface to the PTT must complement those used by the local CO. To maintain compatibility with the local PTT, Intel provides *.cdp* country dependent parameter files that can be modified to satisfy local requirements. User selectable options allow customization of the country dependent parameters to fit a particular application or configuration within a country (for example, switches within the same country may use the same protocol but may require different parameter values for local use). These parameters (for example, the number of DNIS digits, time-outs, party calling number, idle patterns, signaling patterns, and protocol-specific definitions) are specified in the *.cdp* file and may be modified at configuration time (that is, at any

time before starting your application). See the *Global Call Country Dependent Parameters (CDP) Configuration Guide* for additional information.

6.2. Protocol File Naming Conventions

When a protocol is installed on your system, several files are installed, including the protocol module(s), firmware parameter files, and country dependent parameter files. For most protocols, the files are named according to the conventions in *Table 11*.

Table 11. Protocol File Naming Conventions

File Name	Description
ccl_cc_tt_d.hot and *ccl_cc_tt_d.qs*	PDK protocol module (DM3)
ccl_cc_tt_d.psi	PDK protocol module (Springware)
ccl_cc_tt_d.so or *ccl_cc_tt_ffff_d.so*	ICAPI protocol module for Linux (Springware only)
ccl_cc_tt_d.dll or *ccl_cc_tt_ffff_d.dll*	ICAPI protocol module for Windows (Springware only)
ccl_cc_tt_d.cdp or *ccl_cc_tt_ffff_d.cdp*	Country dependent parameter file (DM3 and Springware)

where:

- **ccl** indicates the call control library for which the protocol is written, for example, pdk represents the PDKRT call control library. For the ICAPI call control library, **ccl** is blank.
- **cc** is a 2-character ISO country code, regional code (for example, es = Spain, fr = France, mx = Mexico, na = North America, etc.), or an indication of a switch-specific protocol.
- **tt** is a 2-character protocol type. Valid types are:
  - **em:** a T1 protocol using E&M signaling with support for DTMF digits only

  - **mf:** a T1 protocol using E&M signaling with support for MF digits
  - **r2:** a protocol using R2 MFC signaling
  - **r1:** a protocol using R1 MFC signaling
  - **e1:** a pulse, MF SOCOTEL, or other E1 protocol
  - **sw:** a protocol that is switch specific
  - **ls:** a loop start protocol
- **d** is a 1- or 2-character direction indicator. Valid directions are:
  - **i:** inbound
  - **o:** outbound
  - **io:** inbound/outbound
- **ffff** is optional and defines a special software or hardware feature supported by the protocol; 1 to 4 characters. If the protocol type is "sw", then this field provides additional information about the switch.

  NOTE: Requires ICAPI call control library level 2, or else a compatibility error, EGC_COMPATIBILITY, will be generated when the application attempts to load the protocol.

The protocol name used in the **devicename** parameter of the **gc_OpenEx( )** function is the root name of the *.cdp* file. (On DM3 boards, the protocol is determined at board initialization time and not when a Global Call device is opened. For compatibility, the **gc_OpenEx( )** protocol name may be specified for DM3 boards, but it is not used.)

Most ICAPI protocol releases use separate protocol modules to handle the inbound and the outbound portions of a protocol. For example, *Table 12* describes the files included for the Argentina R2 ICAPI protocol.

Table 12. Sample ICAPI Protocol File Set

Description	Protocol Files	
	Linux	Windows
Inbound protocol module	*ar_r2_i.so*	*ar_r2_i.dll*
Outbound protocol module	*ar_r2_o.so*	*ar_r2_o.dll*
Inbound country dependent parameters	*ar_r2_i.cdp*	*ar_r2_i.cdp*
Outbound country dependent parameters	*ar_r2_o.cdp*	*ar_r2_o.cdp*

PDK protocols are bidirectional protocols. For example, *Table 13* describes the files included with the Argentina R2 PDK protocol.

Table 13. Sample PDK Protocol File Set

Description	Protocol Files Linux and Windows
Bidirectional protocol module (DM3)	*pdk_ar_r2_io.hot, pdk_ar_r2_io.qs*
Bidirectional protocol module (Springware)	*pdk_ar_r2_io.psi*
Bidirectional country dependent parameters (DM3 and Springware)	*pdk_ar_r2_io. cdp*

6.3. Protocol Components

Each protocol requires specific firmware parameter file(s) to be downloaded to the voice and network boards:

- protocol modules
- country dependent parameters (*.cdp*) files
- standard voice and network firmware parameter (*.prm*) files for the country (Springware only)

6.3.1. Protocol Modules

These files contain protocol specific information and are dynamically linked to the application as needed.

PDK protocols are supported on both DM3 and Springware boards. For DM3 boards, the protocol modules are *.hot* and *.qs* files. For Springware boards, the protocol module is a protocol state information (*.psi*) file, a binary file that is interpreted by the PDK run-time component (PDKRT).

ICAPI protocols are supported on Springware boards only. The protocol modules for Linux are *.so* files. The protocol modules for Windows are *.dll* files.

6.3.2. Country Dependent Parameter (.cdp) Files

These files contain country specific and protocol specific parameters for use by Global Call. Country dependent parameter (*.cdp*) files may be customized. Descriptions of the country dependent parameters most likely to be modified for a protocol are provided in the *Global Call Country Dependent Parameters (CDP) Configuration Guide*.

For ICAPI protocols, the special parameter @0 identifies the protocol to be run. This parameter specifies the name of the protocol module (ignoring the filename extension and without the path) to be run by the application. Two variations of the same protocol can be run if two *.cdp* files point to the same protocol module filename after @0.

The *.cdp* file should be located only under the installation directory:

- For Linux: $INTEL_DIALOGIC_CFG
- For Windows: %INTEL_DIALOGIC_CFG%

6.3.3. Parameter (.prm) Files (Springware only)

For some protocols, certain parameters must be specified in the firmware parameter (*.prm*) file to ensure proper operation of the protocol. Refer to the *Global Call Country Dependent Parameters (CDP) Configuration Guide* for any required settings.

The *.prm* files are located:

- For Linux: $INTEL_DIALOGIC_FWL
- For Windows: %INTEL_DIALOGIC_FWL%

7. Debugging Applications

The Global Call debugging utilities are described in this chapter.

NOTE: The information in this chapter is applicable to Springware boards only. For information about the pdktrace tool used with DM3 boards, see the Diagnostics Guide for your system release. The pdktrace tool requires Global Call Protocols Version 4.1 or later.

Global Call includes powerful debugging capabilities for troubleshooting protocol-related problems, including the ability to generate a detailed log file. These debugging tools should not be used during normal operations or when running an application for an extended period of time since they increase the processing load on the system and they can quickly generate a large log file.

NOTE: Only run the debugging and logging utilities on a limited number of channels at a time to avoid the possibility of losing events.

The following sections discuss:

- Debugging applications that use PDK protocols
- Debugging applications that use ICAPI protocols

7.1. Debugging Applications that Use PDK Protocols

The Global Call PDKRT (Protocol Development Kit Run Time) provides a rich set of logging features that are useful to protocol developers and implementers of the engine and call control libraries. The application may add additional log records to the log file when logging is enabled.

7.1.1. Enabling and Disabling the Logging

CAUTION

It is recommended to use logging on an as-needed basis. Logging uses significant resources and can reduce the performance of the Global Call PDKRT call control library. Full logging (debug logging) enabled on many channels can reduce performance to such a degree that time-critical operations are affected and the behavior of a protocol may be altered. The LogView tool is required to view the log file.

The PDKRT call control library provides a service for capturing error and debug information in a log file. Enabling and disabling logging is achieved using the **gc_Start( )** function. Once logging is enabled, the **gc_StartTrace( )** function can be used to enable logging on each individual channel. See *Section 4.22. gc_Start( ) and gc_Stop( )* and *Section 4.23. gc_StartTrace( )* for more information.

The parameters that control the logging mechanism can be set by:

- Populating and using a CCLIB_START_STRUCT. See *Section 7.1.2. Populating and Using a CCLIB_START_STRUCT*.
- Defining the GC_PDK_START_LOG environment variable. See *Section 7.1.3. Defining the GC_PDK_START_LOG Environment Variable*.

When both methods are used, the CCLIB_START_STRUCT takes precedence over the GC_PDK_START_LOG environment variable.

NOTE: Two applications should not use the same log file.

7.1.2. Populating and Using a CCLIB_START_STRUCT

The following code shows an example of how to define a CCLIB_START_STRUCT, populate the fields, and use it to enable logging when issuing the **gc_Start( )** function.

```
GC_START_STRUCT t_GcStart;
CCLIB_START_STRUCT t_PdkStart;
t_PdkStart.cclib_name = “GC_PDKRT_LIB”;
t_PdkStart.cclib_data = "filename: pdktest.log;
```

```
loglevel: ENABLE_DEBUG;
service: R2MF_ENABLE | CAS_ENABLE;
cachedump: WHEN_FULL | THREAD_ON;
channel: B1C1, B2C2-4;
cachesize: 10;
maxfilesize: 0;
mindiskfree: 20";
t_GcStart.num_cclibs = 1;
t_GcStart.cclib_list = (void *)
    (& t_PdkStart);
int t_result = gc_Start((GC_START_STRUCTP)& t_GcStart);
```

NOTE: The example above shows all the possible fields in a **cclib_data** string. In practice, you only need to specify the values of fields that are different than the default values.

The length of the filename must be less than 8 characters.

The value of the **cclib_name** field must be GC_PDKRT_LIB and the **cclib_data** field should have the following format:

```
"field name 1 : field value 1; field name 2 : field value 2; ..."
```

where the allowable field names and values are given in *Table 14*.

Table 14. cclib_data Fields and Values

Field Name	Field Values	Default Value
filename	Log file name	gc_pdk.log
loglevel	See *Table 15*.	ENABLE_FATAL or 5
service	See *Table 16*.	ALL_SERVICES
cachedump	See *Table 17*.	WHEN_FULL or 1
cachesize	Any positive integer	1 (number of records in cache)
channel	See *Table 18*.	B*C*
maxfilesize	Integer	0 (Megabytes)
mindiskfree	Integer	20 (Megabytes)

The fields can be defined in any sequence. If any field is not defined or defined incorrectly (either in name or value), then the default value is used for logging.

The actual values of the fields are posted as the first record of the log file. In this way, when a log file is received, the user knows how logging was configured (that is, which log level and services were enabled, what the cache size and cache dump conditions were when it was generated).

The following examples show how to set the **cclib_data** string:

- The example below shows all the possible fields. In practice, you only have to specify the values of fields that are different than the default values.

```
cclib_data = "filename: pdktest.log;
loglevel: ENABLE_DEBUG;
service: R2MF_ENABLE;
cachedump: WHEN_FULL|THREAD_ON;
channel: B1C1, B2C2-4;
cachesize: 10;
maxfilesize: 0;
mindiskfree: 20"
```

- For simplicity and to avoid errors, use only the values of fields that are different than the default values. For example, to specify a log file name called *mylog.log* that includes all log entries, use the following **cclib_data** string:

```
cclib_data = "filename: mylog.log; loglevel: ENABLE_DEBUG"
```

The following tables show the allowable values for the **loglevel**, **service**, **cachedump**, and **channel** fields respectively. The values of **loglevel**, **service**, and **cachedump** can be numbers or symbols. (If hex format is used, the prefix 0x should be used.) Consequently, before these values are passed to the LOG_INIT, the values must be examined and converted from symbols to numbers, if necessary. The value symbol of **service** and **cachedump** can be a bit mask.

Table 15 shows the valid values for the **loglevel** parameter.

Table 15. Loglevel Parameter Values

loglevel	Valid Value	Description
ENABLE_FATAL (default)	5	Only fatal errors are logged. A fatal error is an error that will make the program run abnormally or will stop the program. For example, in *channelimpl.cpp*, **dx_open( )** returns INVALID_VOICEH. It is expected that an exception will be thrown and the log cache will be dumped to a file if possible.
ENABLE_WARNING	4	All levels above ALERT are logged. An error occurs that may make the program run abnormally. For example, in *channelimpl.cpp*, the new local state is not ChanState_InService while the reason is Wait Call. An exception may be thrown, but log cache will not be dumped to a file automatically.
ENABLE_ALERT	3	All levels above INFO are logged. There is a problem, generally not an error, that the user should know about.
ENABLE_INFO	2	All levels above DEBUG are logged. Important information that the user needs to be aware of is logged. For example, in *channelimpl.cpp*, issuing a **gc_StartTrace( )** and **gc_StopTrace( )** determines if logging for a specific channel is on or off. This kind of information is a level higher than DEBUG.

loglevel	Valid Value	Description
ENABLE_DEBUG	1	All levels are logged. This gives the most detailed information to help debug protocols or code step-by-step. For example, in *channelimpl.cpp*, a call to any of the GC_PDK_C_XXX functions should be logged at this level. Most routine logging should use this level.
Note: Values are in decimal but can also be specified in hex using a 0x prefix.		

Table 16 shows the valid values for the **service** parameter.

Table 16. Service Parameter Values

service	Valid Value	Description
ALL_SERVICES (default)	0xFFFFFFFF (65535)	All services are enabled.
USRAPP_ENABLE	0x00000001 (1)	Only USRAPP service enabled.
GCAPI_ENABLE	0x00000002 (2)	Only GCAPI service enabled.
GCXLTR_ENABLE	0x00000004 (4)	Only GCXLTR service enabled.
LINEADMIN_ENABLE	0x00000008 (8)	Only LINEADMIN service enabled.
CHANNEL_ENABLE	0x00000010 (16)	Only CHANNEL service enabled.
LOADER_ENABLE	0x00000020 (32)	Only LOADER service enabled.
CALL_ENABLE	0x00000040 (64)	Only CALL service enabled.

service	Valid Value	Description
R2MF_ENABLE	0x00000080 (128)	Only R2 MF service enabled.
TONE_ENABLE	0x00000100 (256)	Only TONE service enabled.
CAS_ENABLE	0x00000200 (512)	Only CAS service enabled.
TIMER_ENABLE	0x00000400 (1024)	Only TIMER service enabled.
SDL_ENABLE	0x00000800 (2048)	Only SDL service enabled.
SRL_ENABLE	0x00001000 (4096)	Only SRL service enabled.
ERRHNDLR_ENABLE	0x00002000 (8192)	Only ERRHNDLR service enabled.
LOGGER_ENABLE	0x00004000 (16384)	Only LOGGER service enabled.
RTCM_ENABLE	0x00008000 (32768)	Only RTCM service enabled.
GCLIB_ENABLE	0x00010000 (65546)	Only GCLIB service enabled.
Note: Values prefixed with 0x are hexadecimal values. Decimal values are shown in parentheses.		

Table 17 shows the valid values for the **cachedump** parameter.

Table 17. Cachedump Parameter Values

cachedump	Valid Value	Description
ON_FATAL	0x0000 (bit 1 = 0)	The cache memory will be dumped to the log file once there is a log record with a FATAL level.

cachedump	Valid Value	Description
WHEN_FULL (default)	0x0001 (bit 1 = 1)	The cache memory will be dumped to the log file once the log cache is full as determined by the **cachesize** parameter. For example, if **cachesize** is 10, the log cache is dumped to a file when it contains 10 log records.
THREAD_OFF (default)	0x0000 (bit 2 = 0)	The dump operation will be executed by the calling thread.
THREAD_ON	0x0002 (bit 2 = 1)	The dump operation will be executed by a separate cache dumping thread.
Note: Values prefixed with 0x are hexadecimal values.		

Table 18 shows some examples of the **channel** parameter.

Table 18. Sample Channel Parameter Values

Example Value	Boards and Channels Enabled for Logging
B*C* (default)	All boards and all channels
B-1C-1	Only board number = -1 and channel number = -1.
B1C*	All channels on board 1.
B1C-1	Only board 1 level.
B1C1	Channel 1 on board 1.
B1C1-5	Channels 1 to 5 on board 1.
B1C1,20	Channels 1 and 20 on board 1.
B1-4C*	All channels of boards 1 to 4.
B1C2, B2C2,20-22	Channel 2 on board 1, channels 2, 20, 21, and 22 on board 2.

7.1.3. Defining the GC_PDK_START_LOG Environment Variable

The GC_PDK_START_LOG environment variable can also be used to enable and configure logging.

The following examples show how to set the GC_PDK_START_LOG environment variable in Windows:

- The following is an example of a GC_PDK_START_LOG environment variable definition showing all the possible field values in the environment variable. In practice, you only have to specify the values of fields that are different than the default values.

```
set GC_PDK_START_LOG="filename : pdktest.log;
loglevel: ENABLE_DEBUG; services: ALL_SERVICES;
cachedump : WHEN_FULL | THREAD_ON; channel : B1C1, B2C2-4;
cachesize : 10; maxfilesize : 0; mindiskfree : 20"
```

- For simplicity and to avoid errors, use only the values of fields that are different than the default values. For example, to specify a log file name called *mylog.log* that includes all log entries, use the following GC_PDK_START_LOG environment variable definition:

```
set GC_PDK_START_LOG = "filename: mylog.log; loglevel: ENABLE_DEBUG"
```

  This definition is equivalent to the logging configuration used in *Section 7.1.2. Populating and Using a CCLIB_START_STRUCT* and the definition for each field is also the same as described in that section.

The setting of the environment variable to enable PDK logging in Linux is:

```
export GC_PDK_START_LOG="filename:gc_pdk.log;loglevel:ENABLE_DEBUG;
service:ALL_SERVICES;cachedump:WHEN_FULL|THREAD_OFF;cachesize:1;maxfilesize:2"
```

7.2. Debugging Applications that Use ICAPI Protocols

The parameters shown in *Table 19* are available in the *icapi.cfg* file as debugging tools. Unless otherwise instructed, these parameters should retain their original settings.

The *icapi.cfg* file is located in the following directory:

- For Linux: $INTEL_DIALOGIC_CFG

- For Windows: %INTEL_DIALOGIC_CFG%

When logging is enabled, the log file generated is *icapi.log.<pid>*, where pid = the process identification number.

For Linux applications, the log file is generated by compiling the *country.c* file with the symbol DEBUG defined and then setting the parameters $11 and $12 in the *icapi.cfg* file as indicated in the following table. To write additional information directly to the ICAPI log file, use the **rs_log_printf( )** function. This function works like the **fprintf( )** function except that a file descriptor is not used.

For Windows applications, the log file is generated by setting parameters $11 and $12 in the *icapi.cfg* file as indicated in the following table.

Table 19. icapi.cfg File Parameters

Parameter	Description
$11	Logging utility (default = 0): • Set to 0 to ignore parameters $12, $13 and $15. • Set to 1 to enable logging, either to the screen (set $13 parameter to 1) or to the *icapi.log.<pid>* file to track all the events that occur at the device selected for monitoring (parameter $12). This setting enables the debug tools associated with the protocol. These tools help to locate the source of a protocol problem. **Note:** Enabling logging is not recommended during normal operation due to the increased host processor loading. • (Windows only) Set to 2 to enable logging to a memory buffer and to generate an *icapi.inf* file. The *icapi.inf* file contains the memory address where the debug information is stored.
$12	Number of the channel to be monitored (default = 0): • A value of 0 means monitor all opened devices. • A value of -1 means do not monitor any device. • Entering a channel number designates the channel to be monitored.

Parameter	Description
$13	Echo on screen (default = 0): • Set to 0 to ignore parameter. • Set to 1 to send the debug information to the screen.
$14	Disable DTI Wait Call function (default = 0): • The 0 default value causes the DTI Wait Call firmware function to wait for an incoming call at the board firmware level. • A value of 1 causes the DTI Wait Call firmware function to wait for an incoming call at the ICAPI call control library level. The value selected is protocol-dependent; do not change the default value unless instructed to do so in the documentation for your protocol.
$15	(Linux only) Size of debug memory (default = 1; that is, 1 = 1 event or action in memory) The debug memory saves passed actions or events to a buffer. The built-in debug function does not use this feature. Change this parameter only if you implement your own debug function and you need a larger circular buffer than 1 event or action. • Set to 1 to store one action or event in the buffer. • Set to 0 to ignore feature (default).
$18	Enables cadenced tones, such as ringback and busy, to be played using the firmware rather than using host-based function calls such as **dx_playtone( )** and **sleep( )**. • Set to 0 to disable firmware cadence tones (default). • Set to 1 to enable firmware cadence tones.

Any unspecified parameter defaults to 0. If parameters $13 and $15 are set to 0, they are ignored.

Parameters $16 and $17 (not shown in *Table 19*) are for backwards compatibility only and should not be changed.

Index

F

G

R

S

T

V