Global Call Analog Technology User's Guide

for Linux and Windows

05-1041-008

Publication Date: June 2005

Document Number: 05-1041-008

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 page at: *http://www.intel.com/buy/wtb/wtb1028.htm*

Table of Contents

1. How to Use This Guide 1
1.1. Organization of this Guide 1
1.2. Intel® Dialogic® Products That Support Analog Interfaces 2
1.3. Related Information 2
2. Developing Global Call Analog Loop Start Applications 5
2.1. Analog Telephone Calls 5
2.1.1. Inbound Analog Calls 6
2.1.2. Outbound Analog Calls 7
2.2. Enhanced Call Analysis Concepts 7
2.3. Analog Signaling 8
2.4. Global Tone Detection Considerations 10
2.5. Call Progress and Call Analysis 10
2.5.1. Call Analysis with DM3 Boards 11
2.5.2. Call Analysis for PDKRT Protocols 15
2.6. Supervised Call Transfer 16
2.6.1. Basic Call Transfer Scenario 16
2.6.2. Call Transfer APIs 18
2.6.3. Application Development Notes 18
2.6.4. PBX Testing 19
2.6.5. PBX Integration Issues 20
2.6.6. Configuring the Software 21
2.7. Header Files 22
2.8. Resource Association 22
2.9. Alarm Handling 23
2.10. Network Call Termination 23
2.11. Run Time Configuration of the PDKRT Call Control Library 23
2.12. Run Time Configuration of PDK Protocol Parameters 24
2.13. Determining Protocol Version 27
2.14. Programming Guidelines for PDK Analog Applications 29
3. Applying Global Call Functions to Analog Loop Start Applications 31
3.1. gc_AcceptCall( ) 32
3.2. gc_AnswerCall( ) 32
3.3. gc_Attach( ) and gc_AttachResource( ) 32
3.4. gc_BlindTransfer( ) 33
3.5. gc_Detach( ) 33

3.6. gc_DropCall( ) 33
3.7. gc_GetANI( ) 34
3.8. gc_GetCallInfo( ) 34
3.9. gc_GetParm( ) 35
3.10. gc_MakeCall( ) 35
3.10.1. Use of the timeout Parameter 35
3.10.2. Other gc_MakeCall( ) Considerations 36
3.10.3. PDK_MAKECALL_BLK 37
3.11. gc_OpenEx( ) 38
3.11.1. gc_OpenEx( ) with Springware Boards 38
3.11.2. gc_OpenEx( ) with DM3 Boards 40
3.12. gc_ReleaseCall( ) and gc_ReleaseCallEx( ) 41
3.13. gc_ResetLineDev( ) 41
3.14. gc_SetParm( ) 42
3.15. gc_Start( ) 42
3.16. gc_StartTrace( ) 43
3.17. gc_WaitCall( ) 43

4. Resource Allocation and Routing 45

5. Analog Protocols 47

5.1. Protocols Supported 47
5.2. Protocol File Naming Conventions 48
5.3. Protocol Components 49
5.3.1. Protocol Modules 49
5.3.2. Country Dependent Parameter (.cdp) Files 49

6. Debug Utilities 51

6.1. Enabling and Disabling the Logging 51
6.2. Populating and Using a CCLIB_START_STRUCT 52
6.3. Defining the GC_PDK_START_LOG Environment Variable 58
6.4. Extended Logging 59
6.4.1. gc_ExtensionFunction( ) 59
6.4.2. PDK_XTEN_LOG_FUNC 60
6.4.3. Extended Logging Code Example 61

Index 63

List of Figures

Figure 1. Basic Call Transfer Scenario 17

List of Tables

Table 1. Signaling Used to Dial 9
Table 2. Global Call Call Progress Settings 13
Table 3. Call Analysis Support on DM3 Analog 14
Table 4. Reasons for Network Call Termination 23
Table 5. Configurable PDKRT Call Control Library Parameters 24
Table 6. CDP Parameters 25
Table 7. PSL and SYS Parameters 25
Table 8. Configurable PDK Protocol Parameters 26
Table 9. Analog Call Conditions and Results 36
Table 10. PDK_MAKECALL_BLK Field Descriptions 37
Table 11. Parameters Supported, gc_GetParm( ) and gc_SetParm( ) 42
Table 12. Protocol File Naming Conventions 48
Table 13. PDK North American Analog Protocol File Set 49
Table 14. cclib_data Fields and Values 53
Table 15. Loglevel Parameter Values 54
Table 16. Service Parameter Values 56
Table 17. Cachedump Parameter Values 57
Table 18. Sample Channel Parameter Values 57
Table 19. PDK_XTEN_LOG_FUNC Field Descriptions 60

1. How to Use This Guide

This guide is for users who use the Global Call application programming interface (API) and related software to develop Linux or Windows applications in an analog loop start interface 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 analog loop start 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 analog loop start environment is contained in this *Global Call Analog Technology User's Guide*.

Throughout this document, the terms analog environment, analog loop start, and analog interface refer to the telephone line interface that receives analog voice and telephony signaling information from the telephone network.

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 call control library is identified explicitly.

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. Developing Global Call Analog Loop Start Applications presents guidelines for developing analog loop start applications.

Chapter 3. Applying Global Call Functions to Analog Loop Start Applications describes the additional functionality or the difference in performance of specific Global Call functions when used in an analog loop start environment.

Chapter 4. Resource Allocation and Routing describes using dedicated voice resources in an analog loop start environment.

Chapter 5. Analog Protocols describes the protocol conventions used and programming considerations when incorporating individual country protocol(s) into your application.

Chapter 6. Debug Utilities describes the diagnostic tools available for debugging a Global Call application.

1.2. Intel® Dialogic® Products That Support Analog Interfaces

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 support analog interfaces.

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) for PDK Protocols Configuration Guide* – describes the parameters associated with each of the countries needed for utilizing Global Call.

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.

2. Developing Global Call Analog Loop Start Applications

This chapter offers advice and suggestions for programmers designing and coding Global Call applications in a Linux or Windows environment. Specific guidelines for developing analog loop start applications are provided. Topics include the following:

- Analog telephone calls
- Enhanced call analysis concepts
- Analog signaling
- Global tone detection considerations
- Call progress and call analysis
- Supervised call transfer
- Header files
- Resource association
- Alarm handling
- Network call termination
- Run time configuration of the PDKRT call control library
- Run time configuration of PDK protocol parameters
- Determining protocol version
- Programming guideline for PDK analog applications

2.1. Analog Telephone Calls

For each analog loop start channel, the **gc_OpenEx( )** function is used to open the voice line device and the telephone network interface device (interface to the loop start telephone line or trunk).

For Springware platforms, the **gc_LoadDxParm( )** function is invoked to set voice parameters to be used for the voice channel associated with a line device. These voice parameters are specified in a user-created voice channel parameter (.vcp) ASCII text file. Parameters that are not specified will be assigned their default value automatically. The voice channel parameters include all channel-level parameters set by the voice function, **dx_setparm( ),** and all enhanced call analysis parameters defined in the voice DX_CAP data structure. This feature is not currently available on DM3 platforms.

2.1.1. Inbound Analog Calls

For inbound calls, after the operations described in *Section 2.1. Analog Telephone Calls* complete, a **gc_WaitCall( )** function is issued and waits for an inbound call request on the loop start network interface device.

- When the **gc_WaitCall( )** function is issued synchronously, the function waits for the number of rings defined by the default number of rings parameter (set by the .cdp file—not available on DM3, as calls are offered to the application as soon as the firmware has all of the information needed to present the call) or for time-out to expire. When either condition occurs, the function returns.
- When the **gc_WaitCall( )** function is issued asynchronously, the function completes when an unsolicited GCEV_OFFERED event occurs.

When an inbound call is received, the **gc_AnswerCall( )** function establishes the conditions for answering the call, answers the call, and continues to monitor for a disconnect. The **rings** parameter of the **gc_AnswerCall( )** function is not used, as the analog protocol does not generate the actual ringback; it is generated by the analog switch.

During the call, Global Call continually tests for call disconnect by monitoring for disconnect tones or for a loop current change.

The **gc_CallAck( )** function is not supported for analog calls.

2.1.2. Outbound Analog Calls

For an outbound call, after the operations described in *Section 2.1. Analog Telephone Calls* complete, the **gc_MakeCall( )** function is invoked to make an outgoing call using the specified loop start network and voice resources. First, the channel used to make the outbound call is taken off-hook. Then the number is dialed using DTMF signaling, MF tone signaling, or pulse dialing. (Pulse dialing is not available on DM3 boards.) Call progress tones are monitored to track the progress (current status) of the call. Enhanced call analysis is used for outbound analog telephone calls.

On Springware boards, the call progress tones can be changed from their default values by using the **dx_** API to change the appropriate default tones in the firmware. (Refer to the *Voice API Library Reference*.) Other call analysis parameters can be set by the **gc_LoadDxParm( )** function (Springware only). Global Call analog technology can be configured to not use call progress tones, but the protocol will transition to the connected state immediately after dialing, as there is no way for the protocol to determine connection status with call progress analysis.

2.2. Enhanced Call Analysis Concepts

Intel® Dialogic® analog call technology uses a method of signal identification for call analysis that can also detect fax machines and answering machines.

NOTE: All call analysis parameters (“basic only” and “enhanced”) are supported by Global Call analog call technology.

Call analysis is initiated when a call is dialed. Call parameters are determined by the parameters and values defined in the voice DX_CAP call analysis parameter data structure. On Springware boards, 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. (The default values cannot be changed on DM3 boards.) For a detailed description of enhanced call analysis (PerfectCall) and how to use call analysis, see the *Voice API Programming Guide*.

For each analog 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. The number of tones used and the frequency combinations used to convey this signaling information vary from country to country. Also, whenever a call is switched via networks that do not support or pass caller identification information, then this information can be lost.

The following sections describe analog signaling as it is used in a network, DTMF (dual tone multi-frequency) signaling, global tone detection considerations, and the Global Call call analysis capability.

2.3. Analog Signaling

Analog signaling (DTMF, MF tones, or pulses) transmit the telephone number of the called party to the local CO. 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 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 explicitly identify the called party.

The destination CO checks the called party's line to determine if it is idle or busy. If the called party's line is idle, 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, the destination CO sends this information backwards to the calling party via tones.

NOTE: Analog technology does not provide a means to physically block or unblock an analog line.

Pulse dialing (also called rotary dialing) sends digit information to the CO by momentarily opening and closing (or breaking) the electrical loop from the calling party to the CO. This electrical loop is broken once for the digit 1, twice

for 2, etc., and 10 times for the digit 0. (Pulse dialing is not available on DM3 boards.)

DTMF and MF signaling use a multifrequency code system wherein each DTMF or MF signal is composed of two frequencies, as listed in *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).

Some MF digits use approximately the same frequencies as DTMF digits; for example, the digit 4 uses 770 and 1209 Hz for DTMF 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. Digit detection accuracy depends on the digit sent and 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

Code	Pulse (clicks)	DTMF (Hz)	MF (Hz)
1	1	697, 1209	700, 900
2	2	697, 1336	700, 1100
3	3	697, 1477	900, 1100
4	4	770, 1209	700, 1300
5	5	770, 1336	900, 1300
6	6	770, 1477	1100, 1300
7	7	852, 1209	700, 1500
8	8	852, 1336	900, 1500
9	9	852, 1477	1100, 1500
0	10	941, 1336	1300, 1500

Code	Pulse (clicks)	DTMF (Hz)	MF (Hz)
*	-	941, 1209	1100, 1700
#	-	941, 1477	1500, 1700

2.4. Global Tone Detection Considerations

The Global Call API provides network device independence by shielding the application from protocol-specific details while giving access to each protocol's full range of features.

Since global tone detection (GTD) tones are used for call analysis, the tone definitions are sent to the firmware when the **gc_OpenEx( )** function is issued. The voice channel must be idle. Any pre-existing tones are deleted.

CAUTION

The application must **not** delete tones after the tones are downloaded, or the protocol will fail.

If the application requires additional tones after the initial set of tones are loaded, they must be redefined after calling the **gc_OpenEx( )** function. The tone IDs cannot be in the range from 101-189.

2.5. 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.

Global Call call analysis uses global tone detection (GTD) to detect voice, fax, busy, fast busy, ringback, and Special Information Tones (SIT).

The **gc_MakeCall( )** function defines the maximum time (in seconds) within which a call must be answered. Within that interval, busy and ringback tones can be detected. Global Call will disconnect an outbound call and report a GCEV_CALLSTATUS, GCEV_DISCONNECTED, or GCEV_TASKFAIL event to the application if the call is not answered within the default time-outs defined by the protocol or the **gc_MakeCall( )** function. Global Call can also count the number of rings and report one of these events if the maximum number or rings is reached. The maximum number or rings can be changed by using the **gc_LoadDxParm( )** function (Springware only) to change the Global Call **ca_nbrdna** voice call analysis parameter; otherwise the default value of four rings is used. (The default value cannot be changed on DM3 boards.)

The ringback tone heard on any specific call depends on the specific CO that is serving the called party, not the local CO. The ringback tone must be known in order to complete a call. The ringback tone generates a GCEV_ALERTING event, which is reported to the application.

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 *Section 3.8. gc_GetCallInfo( )* for a list of the values that may be returned when the **info_id** parameter is CONNECT_TYPE.

The following sections discuss:

- Call analysis with DM3 boards
- Call analysis for PDKRT protocols

2.5.1. 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 analog 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_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 DM3 analog. 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 on DM3 analog via the Global Call interface.

Table 3. Call Analysis Support on DM3 Analog

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

Call Analysis Feature	Support on DM3	How Obtained/Notes
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.

2.5.2. Call Analysis for PDKRT Protocols

NOTE: The information in this section is applicable to Springware boards only. DM3 boards do not use PDKRT analog protocols. On DM3, the analog protocol is embedded in the firmware.

The Protocol Development Kit Run-Time (PDKRT) library uses default tones defined in the Intel® Dialogic® Voice library for recognition of call progress tones. Any call progress tone defined by the Voice library will be detected. See the *Voice API Programming Guide* for more information about the default tones and the methods used to change the tones.

PDKRT protocols support call analysis via both the **gc_MakeCall( )** function and two PSL parameters, **PSL_MakeCall_CallProgress** and **PSL_MakeCall_MediaDetect** defined in the .cdp file.

For call progress, when the **PSL_MakeCall_CallProgress** parameter is set to 0, call progress is disabled. When the **PSL_MakeCall_CallProgress** parameter is set to 1, call progress is enabled. When the **PSL_MakeCall_CallProgress** parameter is set to 2, call progress is enabled unless NO_CALL_PROGRESS is

specified in the PDK_MAKECALL_BLK structure used by the **gc_MakeCall( )** function.

For media type detection, when the **PSL_MakeCall_MediaDetect** parameter is set to 1, media type detection is enabled. When the **PSL_MakeCall_MediaDetect** parameter is set to 2, media type detection is disabled unless MEDIA_TYPE_DETECT is specified in the PDK_MAKECALL_BLK structure used by the **gc_MakeCall( )** function. In either case, 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.

2.6. Supervised Call Transfer

NOTE: The information in this section is applicable to DM3 boards only, specifically, to the Intel® Dialogic® DMV160LP Combined Media Board only.

Supervised call transfer is a feature that enables a controller (party A) already in a call with another party (party B) to transfer the call to a third party (party C). The end result is a call between party B and party C. This feature is a common requirement in IVR and voicemail applications.

2.6.1. Basic Call Transfer Scenario

The sequence of events in a supervised call transfer scenario is described below. It is assumed that party A and party B are already in a call.

1. Party A hookflashes party B, placing the call with party B on hold. This call is referred to as the "held" call.
2. Party A dials party C and waits for an answer.
3. Party A notifies party C that the transfer is about to take place. This call is referred to as the "consultation" call.
4. Optionally, party A hookflashes party C and notifies party B of the transfer.
5. Party A hangs up.

6. Parties B and C are connected and the transfer is completed.

The sequence is shown diagrammatically in the following figure.

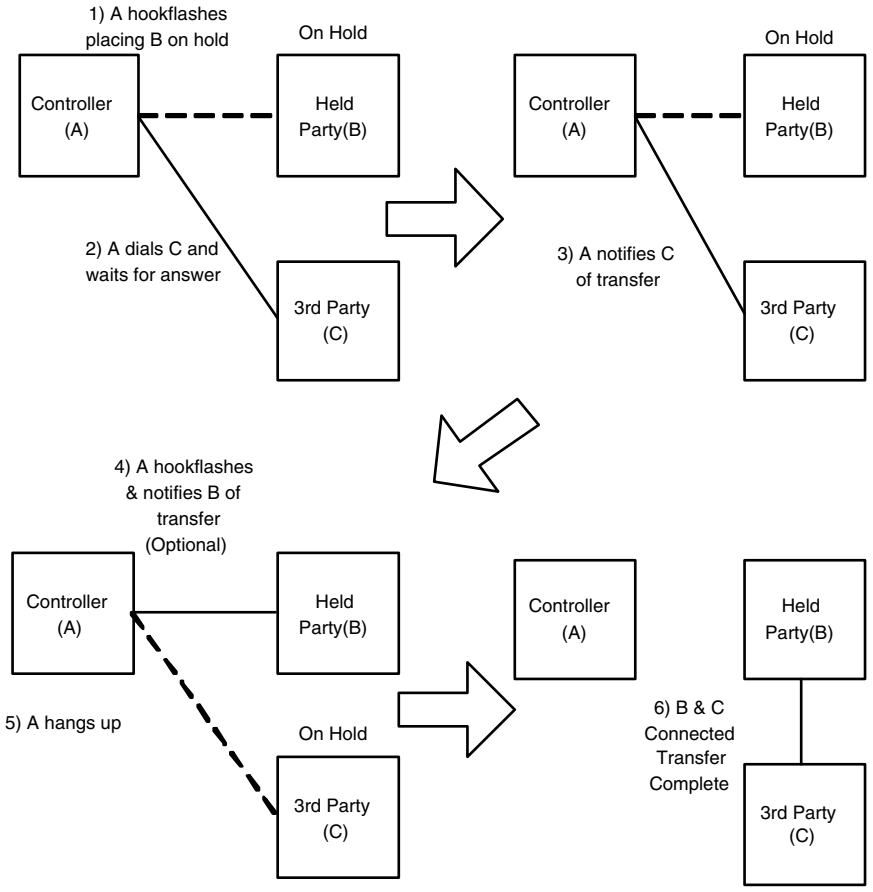

Figure 1. Basic Call Transfer Scenario

2.6.2. Call Transfer APIs

The supervised call transfer feature is provided by the following Global Call API functions:

- **gc_SetupTransfer( )** - initiates a supervised call transfer and allocates a CRN for the consultation call
- **gc_MakeCall( )** - used to make the consultation call
- **gc_CompleteTransfer( )** - used to complete the transfer and communicate to the CPE/CO equipment to connect the talk paths of the held call and the consultation call
- **gc_SwapHold( )** - communicates to the CPE/CO equipment that the talk path to the controller should be "swapped" from the held call to the consultation call. This allows the controller to swap between the Held party and the Third party prior to the transfer. Once this API is completed, the roles of the held and consultation call are reversed.

NOTE: Depending on the PBX type and configuration, it may not be possible to use the **gc_SwapHold( )** function to swap between the held call and the consultation call. For non-US protocols, the **gc_SwapHold( )** function can operate correctly if the behavior of the protocol is similar to that of a US counterpart.

2.6.3. Application Development Notes

The following application development notes apply:

- When any of the parties involved in a transfer are dropped or remotely disconnected prior to calling **gc_CompleteTransfer( )**, all active calls (both consultation and held calls) must be dropped using **gc_DropCall( )** and the CRNs must be released using **gc_ReleaseCallEx( )**.
- The **gc_ResetLineDevice( )** function can be used to reset a channel and terminate all active calls when a transfer call scenario is active.
- When setting up a supervised call transfer, after the **gc_SetupTransfer( )** function is issued to obtain a CRN for the consultation call, a permanent signal timer (8 seconds) starts. If the consultation call is not made within the

8 second period, the timer expires and the application receives a GCEV_DISCONNECTED event.

2.6.4. PBX Testing

NOTE: The call transfer feature has been tested on PBX systems that have been configured to use US protocols only.

The basic call transfer scenario as described above has been tested on the following PBX systems:

- Siemens HiCom 150E Office Pro
- Mitel SX 200
- Ericsson MD110
- Alcatel Omni PCX 4400
- Panasonic Easa-Phone KX T30810
- NEC 2400

For the Siemens HiCom 150E, the following variations in the basic call transfer scenario have also been tested:

- The controller drops the consultation call before dialing is started
  - Party B calls party A
  - Party A hookflashes and then drops the call
  - Verify: Party B is connected back to Party A
- Blind transfer
  - Party B calls party A
  - Party A hookflashes (places call with party B on hold)
  - Party A dials party C, then hangs up
  - Verify: Party B and party C connected
- The held call is dropped by Party B
  - Party B calls party A
  - Party A hookflashes (places call with party B on hold)
  - Party A calls party C (consultation call)
  - Party C picks up
  - Party B hangs up

  - Party C hookflashes
  - Verify: Party C and party A connected
- The consultation call is dropped by Party C
  - Party B calls party A
  - Party A hookflashes
  - Party B calls party C
  - Party C hangs up
  - Verify: Party B and Party A are connected again

2.6.5. PBX Integration Issues

From a PBX perspective, call transfer is most often a sub-feature of Multi-Way Calling (MWC). MWC provides for several variations of conference call capability. Conference features are usually accessed via a flash hook followed by the dialing of an access code. The variation in the behavior of conference features needs to be taken into account when integrating a CT application on a PBX.

The following behavior needs to be considered:

- Swapping between held and consultation calls - If the PBX has conference capability enabled, issuing a second **gc_SwapHold( )** could cause a three-way call to be created. This call scenario can no longer be considered a call transfer scenario.
- Remote party drop of consultation call - There are a number of possible behaviors in this scenario. These include:
  - getting disconnect treatment on party C, or
  - automatically having the talk path connected back to party A (the held call)
- Initiating a call transfer (using **gc_SetupTransfer( )**), then releasing the consultation call prior to issuing **gc_MakeCall( )** - Various types of "ring-back" treatments can be applied by the PBX. A "ring-back" treatment occurs when the PBX generates the ring voltage on any of the parties involved in the transfer (or conference). The duration of the generated ring can be from 1 ring (approximately 6 seconds) to 6 rings (approximately 36 seconds).

2.6.6. Configuring the Software

The following configuration instructions apply when using the DMV160LP board with PBX systems:

- Updating the .config file
- Detecting and learning call progress tones

Updating the .config File

The following parameters require configuration in the DMV160LP .config file:

- **Tone_SigId4** (Disconnect Tone Supervision) must be set to a value of 238113 (a fixed tone ID) to enable disconnect tone supervision. The default value is 0x0 (disabled).
- **BtStartTimeout** (Permanent Signal Planning) must be set to a value appropriate for the PBX system being used. The default is 8000 (8 seconds). This value may need to be changed if the PBX system has a shorter timeout prior to the start of the consultation call.

Whenever a .config file has been modified, a new .fcd file must be generated. This procedure, which must be performed before the board is started, is described in detail in the *Intel DM3 Architecture PCI Products on Windows Configuration Guide*.

Detecting and Learning Call Progress Tones

On Linux, use the learn mode API and tone set file API to detect and learn call progress tones. For more information, see the *Learn Mode and Tone Set File API Software Reference*.

On Windows, use the PBX Expert utility to detect and learn call progress tones.

2.7. Header Files

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 for analog technology:

- *gcpdkrt.h* - required when using PDK error codes, the PDK_MAKECALL_BLK structure for call analysis, or logging via the **gc_Start( )** function.
- *dm3cc_parm.h* - required when developing applications for DM3 platforms; contains the SetIDs and ParmIDs for the different technologies.

2.8. Resource Association

For Springware voice boards with on-board analog loop start devices (for example, D/41ESC, D/160SC-LS), a voice device and an analog loop start device comprise a single channel. Although these devices can be addressed separately, all analog signaling is processed by the associated voice device; analog signaling (ring detection and loop current detection) events are not transmitted over the SCbus. In resource sharing applications using the voice resources of a voice board with on-board analog loop start devices, the analog loop start device associated with a shared voice resource is disabled. See *Chapter 4. Resource Allocation and Routing* for more information.

The Global Call line device ID (LDID) is a single ID that represents the combination of the voice resource and analog loop start (or digital) interface resource that work together to establish and to tear-down calls.

DM3 analog boards are comprised of separate voice devices and analog loop start devices, much like a digital board. As such, these devices are treated separately, with no inherent association between them. Analog loop start devices are denoted as dti devices, just like network time slots are on digital boards. When doing **gc_OpenEx( )** on a DM3 analog device, it is necessary to supply the analog loop start device name as well as a voice device name, or attach a voice device to the analog loop start device after **gc_OpenEx( )**.

2.9. Alarm Handling

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.

The portion of the Global Call call control library that manages alarms, called the Global Call Alarm Management System (GCAMS), is not used. As a result, Global Call applications cannot configure alarm properties and characteristics or receive GCEV_ALARM events.

2.10. Network Call Termination

When a call is terminated by the network, an unsolicited GCEV_DISCONNECTED event is sent to the application. For analog calls, this disconnection may be due to the reasons described in *Table 4. Reasons for Network Call Termination*.

Table 4. Reasons for Network Call Termination

Reason/Message	Global Call Result Value
Disconnect by loop current change	GCRV_NORMAL
Disconnect by tone	GCRV_NORMAL

The application can retrieve the reason for the disconnection using the **gc_ResultInfo( )** function.

2.11. Run Time Configuration of the PDKRT Call Control Library

NOTE: The information in this section is applicable to Springware boards only. DM3 boards do not use PDKRT analog protocols. On DM3, the analog protocol is embedded in the firmware.

Table 5. Configurable PDKRT Call Control Library Parameters 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( )** function 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	CONNECT_TYPE	GCTGT_CCLIB_CRN	Connect type (alternative to **gc_GetCallInfo( )**)	char	GC_R_O
***Note:** GC_R_O - retrieve only					

2.12. Run Time Configuration of PDK Protocol Parameters

NOTE: The information in this section is applicable to Springware boards only. DM3 boards do not use PDKRT analog protocols. On DM3, the analog protocol is embedded in the firmware.

Configurable PDK protocol parameters are grouped in two sets:

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

NOTE: To avoid errors, both 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 shown in *Table 6*.

Table 6. CDP Parameters

Parameter Name	Data Type
CDP_ConnectOnNoRingBack	boolean
CDP_Working_Under_PBX_Env	boolean
CDP_Time_Before_Blind_Dialing_Under_PBX_Env	integer
CDP_Dgts_For_Outside_Line_In_PBX_Env	string
CDP_PBX_DialToneTimeout	integer
CDP_DialTone_As_Disconnect_In_Connected	boolean

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 7*.

Table 7. PSL and SYS Parameters

PSL Variable Name	Data Type
PSL_MakeCall_CallProgress	integer
PSL_MakeCall_MediaDetect	integer
PSL_DefaultMakeCallTimeout	integer
PSL_ANALOG_NUM_RINGS_BEFORE_RINGON	integer
SYS_PSINAME	string

Table 8 shows the Set ID and Parm ID for these parameter types.

Table 8. Configurable PDK Protocol Parameters

Set ID	Parm ID	Target Object Type	Explanation	Update Flag **
PDKSET_ PSI_VAR *	Dynamically assigned	GCTGT_ PROTOCOL_ SYSTEM, GCTGT_PROT OCOL_CHAN	Protocol state information (PSI) variable parameters	GC_W_N
PDKSET_ SERVICE_ VAR	Dynamically assigned	GCTGT_ PROTOCOL_ SYSTEM, GCTGT_PROT OCOL_CHAN	Protocol service layer (PSL) variable parameter and system parameters	GC_W_N
*Indicates that CAS pattern signals and tones cannot be accessed. ** GC_W_N - update only at null state				

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_ConnectOnNoRingBack 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[25] = "CDP_ConnectOnNoRingBack";
   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_ConnectOnNoRingBack" */
   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);
```

2.13. Determining Protocol Version

NOTE: The information in this section is applicable to Springware boards only. DM3 boards do not use PDKRT analog protocols. On DM3, the analog protocol is embedded in the firmware.

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_pdk_na_an_io: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
                    int gc_error;
                    int cclibid;
                    long cc_error;
                    char* gc_msg;
                    char* cc_msg;

                    gc_ErrorValue(&gc_error, &cclibid, &cc_error);
                    gc_ResultMsg(LIBID_GC, (long)gc_error, &gc_msg);
                    gc_ResultMsg(cclibid, cc_error, &cc_msg);
                    printf("gc_GetParm(GCPR_PROTVER) failed! GC(0x%lx) -
                              %s; CC(0x%lx) - %s\n",
```

```
                    gc_error, gc_msg, cc_error, cc_msg);
                    return (gc_error);
                }
            }
        }

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

2.14. Programming Guidelines for PDK Analog Applications

- Because of a limitation in the **dx_** (voice) library, an application using the PDK analog protocol should not call any **dx_** function for a device that is expecting a Global Call termination. For example, when an application calls **gc_OpenEx( )** to open a PDK analog device, the application should not call any **dx_** function on that device before receiving the GCEV_UNBLOCKED event.
- When using the **dx_setevtmsk( )** function, the following **mask** settings **must** be specified for analog channels, in addition to whatever other mask settings are needed in the application:

  ```
  DM_LCOFF|DM_LCON|DM_LCREV|DM_RINGS|DM_RNGOFF|DM_WINK
  ```

  This is because the PDK analog library uses **dx_setevtmsk( )** internally, and needs those masks in order to function correctly. Since every call to **dx_setevtmsk( )** overrides the settings in the previous call, these settings must be included in all calls to **dx_setevtmsk( )** on analog channels, or else the protocol will stop working.

3. Applying Global Call Functions to Analog Loop Start Applications

Certain Global Call functions have additional functionality or perform differently when used in an analog loop start 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 differences in performance of those functions in an analog loop start 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 Global Call analog loop start functions are described in this chapter:

- **gc_AcceptCall( )**
- **gc_AnswerCall( )**
- **gc_Attach( )** and **gc_AttachResource( )**
- **gc_BlindTransfer( )**
- **gc_Detach( )**
- **gc_DropCall( )**
- **gc_GetANI( )**
- **gc_GetCallInfo( )**
- **gc_GetParm( )**
- **gc_MakeCall( )**
- **gc_OpenEx( )**
- **gc_ReleaseCall( )** and **gc_ReleaseCallEx( )**
- **gc_ResetLineDev( )**
- **gc_SetParm( )**
- **gc_Start( )**

- **gc_StartTrace( )**
- **gc_WaitCall( )**

See the *Global Call API Library Reference* for a complete listing of Global Call functions and for detailed function descriptions.

3.1. gc_AcceptCall( )

In the analog protocol, the **rings** parameter is ignored.

The **gc_AcceptCall( )** function provides compatibility only with other Global Call libraries and applications. If your application uses the **gc_AcceptCall( )** function in an analog technology application, calling the function causes an immediate transition to the Accepted state.

The **gc_AcceptCall( )** function is not applicable when using the DMV160LPHIZ board.

3.2. gc_AnswerCall( )

The **gc_AnswerCall( )** function indicates to the remote end that the connection is established (call has been answered). For analog calls, the **rings** parameter of the **gc_AnswerCall( )** function is not used since this value is set by the default number of rings parameter in the .cdp file.

Set the **rings** parameter of the **gc_AnswerCall( )** function to 0 for analog calls.

3.3. gc_Attach( ) and gc_AttachResource( )

NOTE: The **gc_Attach( )** function is deprecated; the preferred equivalent is **gc_AttachResource( )**.

When using analog DM3 boards, if the **gc_Attach( )** or **gc_AttachResource( )** function is issued on a device in the Idle state (for example, between calls), the attach operation is performed, but also an "on-hook" transition occurs in the firmware. This transition is required to allow inbound calls to be answered.

3.4. gc_BlindTransfer( )

The **gc_BlindTransfer( )** function is supported on DM3 analog boards. (The **gc_SetUpTransfer( )**, **gc_CompleteTransfer( )**, and **gc_SwapHold( )** functions, which are used for supervised transfers, are supported only on the DMV160LP board as discussed in S*ection 2.6. Supervised Call Transfer*.)

When used with DM3 analog boards, the **timeout** parameter value of the **gc_BlindTransfer( )** function is ignored.

The **gc_BlindTransfer( )** function is not applicable when using the DMV160LPHIZ board.

3.5. gc_Detach( )

When using analog DM3 boards, if the **gc_Detach( )** function is issued on a device in the Idle state (for example, between calls), the detach operation is performed, but also an "off-hook" transition occurs in the firmware. This transition is required to ensure that no inbound calls will be processed.

3.6. gc_DropCall( )

The **cause** parameter value of the **gc_DropCall( )** function is ignored.

CAUTION

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

The **gc_DropCall( )** function is not applicable when using the DMV160LPHIZ board.

3.7. gc_GetANI( )

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

The **gc_GetANI( )** function only returns the calling party's telephone number (Directory Number, DN). Other information, such as time of day, date, and caller name, may also be available. The **gc_GetCallInfo( )** function can be used to obtain this other information.

The transmission of caller ID information by the CO is protocol dependent. See your protocol in the *Global Call Country Dependent Parameters (CDP) for PDK Protocols Configuration Guide* for required parameter settings.

3.8. gc_GetCallInfo( )

The **gc_GetCallInfo(**) function can be used to retrieve ANI information such as time of day, date, and caller name, if available, from the network. When using this function to retrieve information for an inbound call, the following limitations apply to the **info_id** parameter:

- CALLTIME must be **exactly** AN_MAXCALLTIME bytes in length
- CALLNAME must be **exactly** AN_MAXCALLNAME bytes in length

When using this function to retrieve information for an outbound call, the **info_id** parameter CONNECT_TYPE contains the type of connection as returned by the function. These connection types are:

- GCCT_CAD - connection due to cadence break
- GCCT_LPC - connection due to change in loop current
- 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 applicable

3.9. 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
- On Springware boards:
  - GCPR_CALLPROGRESS

See *Section 3.14. gc_SetParm( )* for more information on the meaning of these parameters.

3.10. gc_MakeCall( )

The **gc_MakeCall( )** function is not applicable when using the DMV160LPHIZ board.

3.10.1. Use of the timeout Parameter

When using voice line devices, the **timeout** argument in the **gc_MakeCall( )** function is supported in both the synchronous and asynchronous programming modes.

If the **timeout** value expires 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*. Also, see the *Global Call Country Dependent Parameters (CDP) for PDK Protocols Configuration Guide* that accompanies your protocol software for other time-outs that may apply to your analog protocol.

If all time-out values are set to 0, no time-out condition will apply.

3.10.2. Other gc_MakeCall( ) Considerations

For analog calls, the dialing mode can be changed by the application by including one of the following case-sensitive dialing codes in the dialing string specified by the **numberstr** parameter:

- P - for pulse mode dialing
- T - for DTMF tone mode dialing
- M - for MF tone mode dialing

When included in the dialing string, the dialing code overrides the default set by the dialing mode parameter in the .cdp file (Springware only). (On DM3, inclusion of these dialing codes causes the digits **not** to be dialed.)

The **gc_MakeCall( )** function description in the *Global Call API Library Reference* provides a table describing call conditions and results. In addition to the information in that table, the values described in *Table 9* apply when running analog technology.

Table 9. Analog Call Conditions and Results

Condition	Event/Return Value	Result/Error Value
No ringback detected	**Async:** GCEV_CALLSTATUS or GCEV_DISCONNECTED	**Async:** GCRV_NORB result value
	Sync: 0	**Sync:** EGC_NORB error
Operator intercept detected	**Async:** GCEV_CALLSTATUS or GCEV_DISCONNECTED	**Async:** GCRV_CEPT result value
	Sync: 0	**Sync:** EGC_CEPT error
Call progress stopped	**Async:** GCEV_CALLSTATUS or GCEV_DISCONNECTED	**Async:** GCRV_STOPD result value
	Sync: 0	**Sync:** EGC_STOPD error
SIT detection error	**Async:** GCEV_CALLSTATUS or GCEV_DISCONNECTED	**Async:** GCRV_CPERROR result value
	Sync: 0	**Sync:** EGC_CPERROR error
No dial tone detected	**Async:** GCEV_DISCONNECTED or GCEV_TASKFAIL	**Async:** GCRV_DIALTONE
	Sync: <0	**Sync:** EGC_DIALTONE

3.10.3. PDK_MAKECALL_BLK

For Springware boards, the PDK_MAKECALL_BLK structure contains information used by the **gc_MakeCall( )** function when setting up a call. 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.

NOTE: Control of call progress and media detection at **gc_MakeCall( )** time works only when the following parameters in the .cdp file are set to allow application control:

```
/* Set to 0 to disable, 1 to enable, and 2 to allow app control */
All INTEGER_t PSL_MakeCall_CallProgress = 2

/* Set 1 to enable, 2 to allow app control */
All INTEGER_t PSL_MakeCall_MediaDetect  = 2
```

The PDK_MAKECALL_BLK structure is defined as follows. See *Table 10* for field descriptions.

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

Table 10. PDK_MAKECALL_BLK Field Descriptions

Field	Description
flags	Contains a bitmask that controls call analysis and media type detection on a per call basis. The possible values that can be ORed are: • NO_CALL_PROGRESS - To disable call analysis. • MEDIA_TYPE_DETECT - To enable media type detection.
*v_rfu_ptr	Reserved for future use.
ul_rfu[4]	Reserved for future use.

Following are some examples:

```
/* To enable Media Detection and disable CPA*/
if (disableCPA && enableMediaDetection)
{
    m_pdkMakecallBlk.flags |= (NO_CALL_PROGRESS|MEDIA_TYPE_DETECT);
    m_gcMakecallBlk.cclib = &m_pdkMakecallBlk;
}

/* To disable CPA */
if (disableCPA)
{
    m_pdkMakecallBlk.flags |= NO_CALL_PROGRESS;
    m_gcMakecallBlk.cclib = &m_pdkMakecallBlk;
}

/* To enable Media Detection */
if (enableMediaDetection)
{
    m_pdkMakecallBlk.flags |= MEDIA_TYPE_DETECT;
    m_gcMakecallBlk.cclib = &m_pdkMakecallBlk;
}
```

3.11. gc_OpenEx( )

The **gc_OpenEx( )** function opens voice channels, voice devices, or analog loop start interfaces. The following sections describe using **gc_OpenEx( )** with Springware boards and with DM3 boards.

3.11.1. gc_OpenEx( ) with Springware Boards

For Springware boards that host both voice devices and analog loop start interface devices, both the voice device and its associated analog loop start interface device are opened as a single channel. A single line device ID (LDID) identifies both the voice channel and the analog loop start interface.

A voice channel, voice device, or analog loop start interface device is specified by the **devicename** parameter using a format that includes the following information:

:P_<protocol_name>:V_<voice_channel_name>

where:

- <**protocol_name**> specifies the analog loop start protocol. Use the root file name of the analog protocol file (for example, *pdk_na_an_io*) for your country or telephone network.
- <**voice_channel_name**> specifies the name of the voice channel, voice device, or analog loop start interface device to be associated with the device being opened. Use the following format for the voice device:

  dxxxB<virtual board number>C<channel or device number>

The prefixes (P_ and V_) in **devicename** are used for parsing purposes. The order of input of these parameters may be set by the application. The fields within the **devicename** parameter must each begin with a colon.

The following example illustrates the format for defining the **devicename** parameter for voice and analog loop start interface devices when processing analog calls.

To open voice channel 2 on a D/160SC-LS board identified as virtual board 3:

:P_pdk_na_an_io:V_dxxxB3C2

Global Call automatically opens both voice device 2 and analog loop start interface device 2 on virtual board 3 and internally attaches the voice device to the analog loop start interface.

NOTE: When using analog protocol, opening a board device, that is, using only the board number in the **devicename** parameter (for example, :V_dxxxB1) is not supported.

3.11.2. gc_OpenEx( ) with DM3 Boards

On DM3 analog boards, the voice devices and the analog loop start devices are separate devices. When calling **gc_OpenEx( )**, the application can either open only the analog loop start device, or open both the analog loop start interface device and the voice device at the same time. The device(s) is specified via the **devicename** parameter using a format that includes the following information:

:P_<protocol_name>:N_<loop_start_device_name>:V_<voice_channel_name>

where:

- <**protocol_name**> can be any string. :P_ must be present, but the name of the protocol does not matter, as the protocol is embedded in the firmware and cannot be changed.
- <**loop_start_device_name**> specifies the name of the analog loop start device to be associated with the device being opened. Use the following format for the loop start device:

  dtiB<virtual board number>T<channel or device number>

- <**voice_channel_name**> specifies the name of the optional voice channel to be associated with the device being opened. Use the following format for the voice channel name:

  dxxxB<virtual board number>C<channel or device number>

The prefixes (P_, N_, and V_) in **devicename** are used for parsing purposes. The order of input of these parameters may be set by the application. The fields within the **devicename** parameter must each begin with a colon.

The following example illustrates the format for defining the **devicename** parameter for voice and analog loop start interface devices when processing analog calls.

To open loop start device 2 on a DMV160 board identified as virtual board 1 along with voice device 1 on virtual board 2:

:P_dm3an:N_dtiB1T2:V_dxxxB2C1

Global Call automatically opens both the analog loop start interface and voice device and internally attaches the voice device to the analog loop start interface.

The voice device could be opened later using the **dx_open( )** function and then attached to the Global Call device using the **gc_AttachResource( )** function. The application would have to manually route the resources together. Without a voice device attached to the Global Call device, no outbound calls can be made as there will be no resources to dial digits.

3.12. gc_ReleaseCall( ) and gc_ReleaseCallEx( )

NOTE: The **gc_ReleaseCall( )** function is deprecated; the preferred equivalent is **gc_ReleaseCallEx( )**.

The **gc_ReleaseCallEx( )** function must be called after a **gc_DropCall( )** function completes. If a new inbound call has arrived since the last **gc_DropCall( )** function was issued, that call will be pending until the **gc_ReleaseCallEx( )** function is called.

If a **gc_WaitCall( )** function is issued asynchronously, the inbound call notification can be received immediately after the **gc_ReleaseCallEx( )** function is called. If a **gc_WaitCall( )** function is issued synchronously and a **gc_ReleaseCallEx( )** function is issued subsequently, the inbound call will be pending until the **gc_WaitCall( )** function is issued again.

The **gc_ReleaseCall( )** and **gc_ReleaseCallEx( )** functions are not applicable when using the DMV160LPHIZ board.

3.13. gc_ResetLineDev( )

The **gc_ResetLineDev( )** function is used to ensure that voice channels are set on-hook. The **gc_ResetLineDev( )** function also sets the Global Call call state to Idle. Placing each voice channel on-hook ensures that any active calls are disconnected and eliminates the possibility of leaving a line in a ringing condition.

The **gc_ResetLineDev( )** function can be called only in the asynchronous mode. You must wait until the GCEV_RESETLINEDEV event is received from each

voice channel before continuing to ensure that the voice channels have been set on-hook.

3.14. 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 11* are supported.

Table 11. 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_MEDI ADETECT	channel	Enables or disables post-connect call progress or media detection; disabled by default.	DM3

For further information about the use of the GCPR_CALLPROGRESS and GCPR_MEDIADETECT parameters, see *Section 2.5.1. Call Analysis with DM3 Boards*.

NOTE: The **gc_SetParm( )** function is not supported when using PDK analog.

3.15. gc_Start( )

For PDK protocols, the **gc_Start( )** function is used to access the error and debug logging capabilities of the PDKRT call control library. See *Chapter 6. Debug Utilities* for more information.

3.16. gc_StartTrace( )

For 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 CCLIB_START_STRUCT data structure. See *Section 6.2. Populating and Using a CCLIB_START_STRUCT* for more information.

3.17. gc_WaitCall( )

The gc_WaitCall( ) function is not applicable when using the DMV160LPHIZ board.

4. Resource Allocation and Routing

Analog loop start protocols require a voice or tone resource for setting up a call. Application development considerations for using dedicated voice resources in an analog loop start environment are discussed in this chapter.

For Springware boards with on-board analog loop start devices, a voice device and an analog loop start device comprise a single channel. Although these devices can be addressed separately, all analog signaling is processed by the associated voice device; analog signaling (ring detection and loop current detection) events are not transmitted over the SCbus. For DM3 boards with on-board analog loop start devices, the voice device and the analog loop start device are separate devices.

Applications requiring voice resources during the entire call (for example, voice-mail, announcements) must have enough voice channels to dedicate one channel to each analog loop start channel. A single **gc_OpenEx( )** function call can open both the analog loop start device and voice device on both Springware and DM3.

To perform activities such as routing and voice store and forward, use the **gc_GetVoiceH( )** functions to obtain the voice handle associated with a line device. For example, before playing a file, you can retrieve the voice handle using the **gc_GetVoiceH( )** function. If needed, you may route other resources to the analog loop start channel (for example, to send a fax) and reroute the voice channel back to the analog loop start channel before setting up or waiting for another call. You must route the same voice channel back to the associated analog loop start channel on Springware because these two resources were internally attached when opened. On DM3, no such restriction exists because the two resources are independent of each other.

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

```
/* Open a Global Call device with a voice channel and an
    analog loop start network time slot */
if (gc_OpenEx(&linedev, ":P_pdk_na_an_io:V_dxxxB1C1", 0, &usrattr)
          == EGC_NOERR) {
        /*
         * Wait for GCEV_UNBLOCKED event.
         */
        .
        .
        .
        /* Make an outgoing call */

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

1

2

Legend:

1 The **gc_OpenEx( )** function:

- opens a Global Call line device using voice channel dxxxB1C1 and configures the line device to use North American Analog Protocol.
- opens the analog loop start time slot and voice channel automatically

SCbus time slot routing and attaching are done automatically. The function need only be called once for an analog loop start time slot/voice channel pair.

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

5. Analog Protocols

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

NOTE: The information in this chapter is applicable to Springware boards only. On DM3 boards, the analog protocol is embedded in the firmware.

5.1. 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. With newer Intel® Dialogic® system releases, the protocol package is included with the system release software.

The Global Call protocols available are listed in the *Global Call Country Dependent Parameters (CDP) for PDK Protocols 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 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 are specified in the country dependent parameter (.cdp) file and may be modified at configuration time (that is, at any time before starting your application). The *Global Call Country Dependent Parameters (CDP) for PDK Protocols Configuration Guide* that accompanies the protocol software lists each supported protocol and describes the modifiable parameters in the protocol's .cdp file.

When using PDK protocols, some parameters are dynamically updateable (that is, the parameter value can be changed while the application is running). See *Section 2.12. Run Time Configuration of PDK Protocol Parameters* for more information.

5.2. Protocol File Naming Conventions

When a protocol is installed on your system, several files are installed, including the protocol module(s) and country dependent parameter files. The Global Call analog loop start protocol files use the naming conventions described in *Table 12*.

Table 12. Protocol File Naming Conventions

Filename	Description
pdk_cc_tt_d.psi	PDK protocol state information file
pdk_cc_tt_d.cdp or *pdk_cc_tt_ffff_d.cdp*	country dependent parameter file

where:

- **pdk** is a prefix for PDK protocols only.
- **cc** is a 2-character ISO country code, for example, na = North America.

  NOTE: The country code na is used to designate protocols used in both the United States and Canada.
- **tt** is a 2-character protocol type, for example, an = analog 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.

The protocol name used in the **devicename** parameter of the **gc_OpenEx( )** function is the root name of the .cdp file (for example, *pdk_na_an_io* for North America).

Examples of the files included for the PDK North American analog protocol are listed in *Table 13*.

Table 13. PDK North American Analog Protocol File Set

Description	Protocol Files Linux and Windows
Analog loop start protocol module	*pdk_na_an_io.psi*
Inbound/outbound country dependent parameters	*pdk_na_an_io.cdp*

5.3. Protocol Components

The file types included with a protocol are:

- protocol modules
- country dependent parameter (.cdp) files

5.3.1. Protocol Modules

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

For PDK, the protocol module is a protocol state information (.psi) file, a binary file that is interpreted by the PDK run-time component (PDKRT).

5.3.2. Country Dependent Parameter (.cdp) Files

In addition to the voice parameter file loaded by the **gc_LoadDxParm( )** function, Global Call uses a country dependent parameter (.cdp) file that defines country specific and protocol specific parameters. For some protocols, certain parameters must be set in the country dependent parameter file to ensure proper operation of the protocol. Country dependent parameter files may be customized by modifying the file using any utility or word processor that can edit and save ASCII text. Refer to the *Global Call Country Dependent Parameters (CDP) for PDK Protocols Configuration Guide* or the Release Note for your analog protocol for country dependent parameters that are most likely to be modified and for any required settings.

6. Debug Utilities

The Global Call debugging utilities are described in this chapter.

NOTE: The information in this chapter is applicable to Springware boards only. There are no specific protocol debugging capabilities for DM3 boards. Debugging of DM3 applications can usually be done with RTF logs.

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:

- Enabling and disabling the logging
- Populating and Using a CCLIB_START_STRUCT
- Defining the GC_PDK_START_LOG environment variable
- Extended logging

6.1. Enabling and Disabling the Logging

In a Springware environment, 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.

CAUTION

It is recommend that logging be done 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 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.

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

- Populating and using a CCLIB_START_STRUCT. See *Section 6.2. Populating and Using a CCLIB_START_STRUCT*.
- Defining the GC_PDK_START_LOG environment variable. See *Section 6.3. 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.

6.2. 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;
binaryfile: 1;
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 value of the **cclib_name** field must be either GC_PDKRT_LIB or PDGV_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 or 0xFFFFFFFF
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, and 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;
binaryfile: 1;
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.

loglevel	Valid Value	Description
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.
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.
R2MF_ENABLE	0x00000080 (128)	Only R2MF 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.
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	Channel 1 to 5 on board 1

Example Value	Boards and Channels Enabled for Logging
B1C1,20	Channel 1 and 20 on board 1
B1-4C*	All channels of boards 1 to 4
B1C2, B2C2,20-22	Channel 2 on board 1, channel 2, 20, 21, and 22 on board 2

6.3. 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.

- The following is an example of a GC_PDK_START_LOG environmental 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; binaryfile : 1;
loglevel: ENABLE_DEBUG; service : R2MF_ENABLE | CAS_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 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 6.2. Populating and Using a CCLIB_START_STRUCT* and the definition for each field is also the same as described in that section.

6.4. Extended Logging

The **gc_ExtensionFunction( )** function provides extended features directly from the call control libraries. For applications that use the PDK protocols, if logging is enabled, the **gc_ExtensionFunction( )** function can be used to add user-specified log records to the log file.

6.4.1. gc_ExtensionFunction( )

For debugging purposes, the **gc_ExtensionFunction( )** should only be used if requested by Intel Technical Support. It enables users to include debug information useful to Technical Support personnel when reading the log file. The log file is a binary file that cannot be read without the required tools, which are supplied with the Protocol Development Kit Run-Time (PDKRT).

The function header of the **gc_ExtensionFunction( )** function is:

```
gc_ExtensionFunction(int cclibid, LINEDEV linedev, CRN crn,
                     void *datap)
```

where:

- **cclibid** is the Global Call call control library ID
- **linedev** is the Global Call line device handle
- **crn** is the call reference number
- **datap** is a pointer to a call control library-specific structure containing information about the extended feature

For extended logging, the **datap** parameter is a pointer to a structure of type PDK_XTEN_LOG_FUNC, which contains extended logging information. See *Section 6.4.2. PDK_XTEN_LOG_FUNC* for more information.

6.4.2. PDK_XTEN_LOG_FUNC

For extended logging, the **gc_ExtensionFunction( )** uses the PDK_XTEN_LOG_FUNC data structure. The structure definition is as follows:

```
typedef struct
    {
        PDK_XTEN_FUNCNUM     func_no;
        char*                log_data;
        PDK_LOG_LEVEL        log_level;
        PDK_SERVICE          service;
        char*                file_name;
        long                 line_num;
    } PDK_XTEN_LOG_FUNC;
```

Table 19 describes each field in the data structure.

Table 19. PDK_XTEN_LOG_FUNC Field Descriptions

Field	Description
func_no	Identifies the extension feature requested. Possible values are: • PDK_FUNC_LOG = 1 • PDK_FUNC_DUMPLOG = 2
log_data	A string that is to be added to the log file.
log_level	The logging level of the added record. Valid logging levels are: • PDK_LOGLEVEL_DEBUG = 1 • PDK_LOGLEVEL_INFO = 2 • PDK_LOGLEVEL_ALERT = 3 • PDK_LOGLEVEL_WARNING = 4 • PDK_LOGLEVEL_FATAL = 5

Field	Description
service	The service name. Valid values are: • PDK_SERVICE_USRAPP = 1 • PDK_SERVICE_GCAPI = 2
file_name	The name of the source file from which the log entry originated.
line_num	The line number in the source file from which the log entry originated.

6.4.3. Extended Logging Code Example

The following code example shows how to include user-defined log records in the log file.

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

void main()
    {
        GC_START_STRUCT gc_start;
        PDK_START_STRUCT pdk_start;
        PDK_XTEN_LOG_FUNC logstruct;
        char *data = "This is a log record";
        LINEDEV ldev;

        pdk_start.cclib_name = "PDKRT";
        pdk_start.start_parameters = "filename: pdkrt;
        loglevel: ENABLE_DEBUG";

        gc_start.nStartStructures = 1;
        gc_start.cclib_start_struct[0] =
             (CCLIB_START_STRUCTP)&pdk_start;

        gc_Start(&gc_start);
        gc_Open(&ldev, ":N_dtiB1T1:P_us_t1_em:" ,0);

        logstruct.func_no = PDK_FUNC_LOG;
        logstruct.log_data = data;
        logstruct.log_level = PDK_LOGLEVEL_ALERT;
        logstruct.service = PDK_SERVICE_USRAPP;
```

```
        logstruct.file_name = __FILE__;
        logstruct.line_num = __LINE__;
        gc_ExtensionFunction(PDGV_LIB,ldev,0,&logstruct);
        /* PDGV_LIB is the ID of the PDKRT */
        /* the rest of the application goes here */
    }
```

Index

.

.cdp file, 47

A

additional tones, 10

alarm handling, 22

analog loop start, 5

analog loop start device, 22, 45

analog signaling, 8, 22, 45

ANI information, 34

answering machine
    detect, 7

answering machine detection, 34

application
    designing and coding, 5

audio tones, 8

B

block analog line, 8

C

cadence break, 34

call analysis, 7, 10

call disconnect, 6

call parameters, 7

call progress, 10

call progress tones, 7

call termination
    network, 23

call transfer, 16

called party, 8

caller ID, 34

calling party, 8

cause parameter, 33

CCLIB_START_STRUCT
    using for debugging, 52

channel-level parameters, 6

code example
    extended logging, 61

connection types, 34

country dependent parameter, 47

D

debugging, 51
    cclib_data fields and values, 53
    enabling for PDK protocols, 52
    log_cachedump values, 57
    log_level values, 54
    log_service values, 55
    PDK protocols, 51
    sample log_channel values, 57

dedicated voice resources, 45
    example, 45

delete tones, 10

destination CO, 7, 8

devicename, 38, 40

devicename parameter, 48

dialed digits, 8

dialing code

case-sensitive, 36
dialing mode, 36
digit detection accuracy, 9
directory number
DN, 34
disconnect tones, 6
disconnection
reason, 23
DN
directory number, 34
DTMF digits, 9
DTMF signaling, 7, 9
DX_CAP data structure, 6, 7
dx_setevtmsk( ), 29
dx_setparm( ), 6

E

enhanced call analysis, 7
enhanced call analysis parameters, 6
event mask settings, 29
extended logging, 59
code example, 61

F

fax machine
detect, 7
fax machine detection, 34
frequency overlap, 9

G

gc_AcceptCall( ), 32
gc_AnswerCall( ), 6, 32
gc_Attach( ), 32
gc_AttachResource( ), 32
gc_BlindTransfer( ), 33
gc_CallAck( ), 6
gc_CompleteTransfer( ), 18
gc_Detach( ), 33
gc_DropCall( ), 33, 41
gc_ExtensionFunction( ), 59
gc_GetANI( ), 34
gc_GetCallInfo( ), 12, 14, 34
gc_GetParm( ), 35
gc_GetVoiceH( ), 45
gc_LoadDxParm( ), 6, 7, 49
gc_MakeCall( ), 17
gc_MakeCall( ), 7, 12, 35, 46
gc_Open( ), 46
gc_OpenEx( ), 5, 38, 45, 48
gc_ReleaseCall( ), 41
gc_ReleaseCallEx( ), 41
gc_ResetLineDev( ), 41
gc_ResultValue( ), 12, 14
gc_SetParm( ), 12, 42
gc_SetupTransfer( ), 17
gc_Start( ), 42
gc_StartTrace( ), 43
gc_SwapHold( ), 18
gc_WaitCall( ), 6, 41, 43
GCEV_ALERTING, 11

GCEV_DISCONNECTED, 23

GCEV_OFFERED, 6

GCEV_RESETLINEDEV, 41

GCPR_CALLPROGRESS, 12, 42

GCPR_MEDIADETECT, 12, 14, 42

global tone detection, 10

GTD, 10, 11

I

inbound call, 6, 8, 41

info_id parameter, 34

international networks, 9

L

LDID
    line device ID, 22

line device ID
    LDID, 22

local CO, 7, 8, 47

log file, 51

logging
    extended, 59

loop current, 34

loop current change, 6

loop current detection
    analog signaling, 22, 45

M

MF digits, 9

MF signaling, 9

MF tone signaling, 7

multifrequency code, 9

multifrequency combinations, 9

N

naming convention
    protocol, 48

national networks, 9

network device independence, 10

network handle, 45

number of rings, 6

numberstr parameter
    dialing string, 36

O

options
    protocol, 47

outbound call, 7, 8

P

PDK protocols
    debugging, 51
    enabling debugging, 52
    programming guideline, 29

PDK_MAKECALL_BLK, 37

PDK_XTEN_LOG_FUNC, 60

PDKRT protocols
    call analysis, 15

PerfectCall, 7

pre-existing tones, 10

protocol, 47
    naming convention, 48
    service layer parameters, 25
    troubleshooting, 51

protocol module, 49
    ICAPI, 24
    PDK, 24

pulse dialing, 7, 8

R

remote end, 32

resource sharing, 22

ring detection
    analog signaling, 22, 45

ringback tone, 8, 11

rings parameter, 6, 32

rotary dialing, 8

S

SCbus, 46

service layer parameters, 25

setting up a call, 37

signaling frequencies, 9

signaling information, 7

SIT, 11
    special information tones, 11

special information tones
    SIT, 11

supervised call transfer, 16

T

telephone number
    called party, 8

timeout, 35

tone definition, 10

tone ID, 10

tone resource, 45

tones downloaded, 10

troubleshooting, 51

U

unblock analog line, 8

V

voice channel, 10

voice channel parameter (.vcp)
    ASCII text file, 6

voice detection, 34

voice device, 45

voice handle, 45

voice resource, 45
    dedicated, 45