

GlobalCall™ Analog Technology User's Guide for Linux and Windows

Copyright © 2001 Dialogic Corporation

05-1041-005

COPYRIGHT NOTICE

All contents of this document are subject to change without notice and do not represent a commitment on the part of Dialogic Corporation. Every effort is made to ensure the accuracy of this information. However, due to ongoing product improvements and revisions, Dialogic Corporation cannot guarantee the accuracy of this material, nor can it accept responsibility for errors or omissions. No warranties of any nature are extended by the information contained in these copyrighted materials. Use or implementation of any one of the concepts, applications, or ideas described in this document or on Web pages maintained by Dialogic may infringe one or more patents or other intellectual property rights owned by third parties. Dialogic does not condone or encourage such infringement. Dialogic makes no warranty with respect to such infringement, nor does Dialogic waive any of its own intellectual property rights which may cover systems implementing one or more of the ideas contained herein. Procurement of appropriate intellectual property rights and licenses is solely the responsibility of the system implementer. The software referred to in this document is provided under a Software License Agreement. Refer to the Software License Agreement for complete details governing the use of the software.

All names, products, and services mentioned herein are the trademarks or registered trademarks of their respective organizations and are the sole property of their respective owners. DIALOGIC (including the Dialogic logo), DTI/124, and SpringBoard are registered trademarks of Dialogic Corporation. A detailed trademark listing can be found at: <http://www.dialogic.com/legal.htm>.

Publication Date: September, 2001

Part Number: 05-1041-005

Dialogic, an Intel Company
1515 Route 10
Parsippany NJ 07054
U.S.A.

For **Technical Support**, visit the Dialogic support website at:
<http://support.dialogic.com>

For **Sales Offices** and other contact information, visit the main Dialogic website at:
<http://www.dialogic.com>

Table of Contents

| | |
|---|-----------|
| 1. How to Use This Guide | 1 |
| 1.1. Dialogic Products That Support Analog Interfaces | 1 |
| 1.2. Organization of this Guide | 1 |
| 2. Developing GlobalCall Analog Loop Start Applications | 3 |
| 2.1. Analog Telephone Calls | 3 |
| 2.1.1. Inbound Analog Calls | 4 |
| 2.1.2. Outbound Analog Calls | 4 |
| 2.2. Enhanced Call Analysis Concepts | 5 |
| 2.2.1. Analog Signaling | 5 |
| 2.3. Global Tone Detection Considerations | 7 |
| 2.4. GlobalCall Call Analysis Capability | 8 |
| 2.4.1. Call Analysis for ANAPI Protocols | 9 |
| 2.4.2. Call Analysis for PDKRT Protocols | 10 |
| 2.5. Header Files | 11 |
| 2.6. Resource Association | 11 |
| 2.7. Alarm Handling..... | 12 |
| 2.8. Network Call Termination..... | 12 |
| 2.9. Run Time Configuration of the PDKRT Call Control Library | 12 |
| 2.10. Run Time Configuration of PDK Protocol Parameters | 13 |
| 2.11. Determining Protocol Version..... | 16 |
| 3. Applying GlobalCall Functions to Analog Loop Start Applications | 19 |
| 3.1. gc_AcceptCall() | 19 |
| 3.2. gc_AnswerCall() | 20 |
| 3.3. gc_DropCall() | 20 |
| 3.4. gc_GetANI() | 20 |
| 3.5. gc_GetCallInfo() | 21 |
| 3.6. gc_MakeCall()..... | 21 |
| 3.7. gc_OpenEx() | 23 |
| 3.8. gc_ReleaseCallEx()..... | 24 |
| 3.9. gc_ResetLineDev()..... | 24 |
| 3.10. gc_Start()..... | 24 |
| 3.11. gc_StartTrace()..... | 25 |
| 4. Resource Allocation and Routing | 27 |
| 5. Analog Protocols | 29 |

GlobalCall™ Analog Technology User's Guide for Linux and Windows

| | |
|--|-----------|
| 5.1. Protocol Naming Convention | 30 |
| 5.2. Protocol Components | 32 |
| 5.2.1. Country Dependent Parameter (.cdp) Files | 32 |
| 5.2.2. Site Dependent Parameter (.sdp) Files | 33 |
| 6. Debug Utilities | 35 |
| 6.1. Debugging Applications that use ANAPI Protocols | 35 |
| 6.2. Debugging Applications that use PDK Protocols | 37 |
| 6.2.1. Enabling and Disabling the Logging | 37 |
| 6.2.2. Extended Logging | 44 |
| 6.2.3. gc_ExtensionFunction() | 44 |
| 6.2.4. PDK_XTEN_LOG_FUNC | 45 |
| 6.2.5. Extended Logging Code Example | 46 |
| Index | 49 |

List of Tables

| | |
|--|----|
| Table 1. Signaling Used to Dial | 7 |
| Table 2. Reasons for Network Call Termination..... | 12 |
| Table 3. Configurable PDKRT Call Control Library Parameters..... | 13 |
| Table 4. CDP Parameters | 14 |
| Table 5. PSL and SYS Parameters | 14 |
| Table 6. Configurable PDK Protocol Parameters..... | 15 |
| Table 7. Analog Call Conditions and Results..... | 22 |
| Table 8. Protocol File Naming Conventions | 30 |
| Table 9. Protocol Component Names..... | 30 |
| Table 10. Sample ICAPI Protocol File Set..... | 31 |
| Table 11. Sample PDK Protocol File Set | 31 |
| Table 12. anapi.cfg File..... | 36 |
| Table 13. cclib_data Fields and Values | 38 |
| Table 14. log_level Values..... | 40 |
| Table 15. log_service Values | 41 |
| Table 16. log_cachedump Values | 42 |
| Table 17. Sample log_channel Values | 43 |
| Table 18. PDK_XTEN_LOG_FUNC Field Descriptions | 45 |

GlobalCall™ Analog Technology User's Guide for Linux and Windows

1. How to Use This Guide

This guide is for users who the GlobalCall™ Application Programming Interface (API) and related software to develop Linux or Windows applications in an analog loop start interface environment. Use this guide in conjunction with the *GlobalCall API Software Reference* and the *GlobalCall Country Dependent Parameters (CDP) Reference* for the protocols supported by your application. Products covered by this guide and the organization of this guide are described in this chapter. 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.

Differences between the implementation of a GlobalCall application in a Linux or a Windows environment are either described parenthetically or are presented in separate paragraphs/sections.

1.1. Dialogic Products That Support Analog Interfaces

The GlobalCall software provides a consistent interface across Dialogic products interfaced to various networks (for example, Analog, E-1 CAS, T-1 robbed bit ISDN, and others). See the *Release Guide* for your operating system for the Dialogic product combinations that support Analog interfaces.

1.2. Organization of this Guide

This guide provides information for developing analog loop start GlobalCall API applications and details differences in GlobalCall function usage in analog loop start applications as follows:

Chapter 2 presents guidelines for developing Analog loop start applications.

Chapter 3 describes the additional functionality of specific GlobalCall functions used for developing Analog loop start applications.

Chapter 4 describes using dedicated voice resources in an analog loop start environment.

GlobalCall™ Analog Technology User's Guide for Linux and Windows

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

Chapter 6 describes the debugging capabilities for troubleshooting.

An **Index** follows the last chapter.

2. Developing GlobalCall Analog Loop Start Applications

This chapter offers advice and suggestions for programmers designing and coding GlobalCall 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
- Global Tone Detection Considerations
- GlobalCall Call Analysis Capability
- 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

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).

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.

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) 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 since this value was previously set by the default number of rings parameter in the *.cdp* file.

During the call, GlobalCall 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. Call progress tones are monitored to track the progress (current status) of the call. Enhanced call analysis is used for outbound analog telephone calls.

The call progress tones can be changed from their default values by editing the *.cdp* file. Other call analysis parameters can be set by the **gc_LoadDxParm()** function. GlobalCall analog technology always uses call progress tones.

2. Developing GlobalCall Analog Loop Start Applications

2.2. Enhanced Call Analysis Concepts

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 GlobalCall 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. The default parameter values defined in the DX_CAP data structure can be changed by the **gc_LoadDxParm()** function to fit the needs of your application. For a detailed description of enhanced call analysis (PerfectCall) and how to use call analysis, see the *Call Analysis Chapter* in the *Voice Software Reference – Features Guide* for your operating system.

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 paragraphs describe analog signaling as it is used in a network, DTMF (Dual Tone Multi-Frequency) signaling, Global Tone Detection considerations, and the GlobalCall call analysis capability. See also *Section 5.2.1. Country Dependent Parameter (.cdp) Files* for using call progress tones to determine the condition of a call.

2.2.1. 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.

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 *DTMF and MF Tone Specifications Appendix* in the *Voice Software Reference - Programmer's Guide* for your operating system for details.

2. Developing GlobalCall Analog Loop Start Applications

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 |
| * | - | 941, 1209 | 1100, 1700 |
| # | - | 941, 1477 | 1500, 1700 |

2.3. Global Tone Detection Considerations

The GlobalCall 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.

2.4. GlobalCall Call Analysis Capability

GlobalCall uses the Call Analysis capability of the voice and analog interface resources to monitor the progress of an outbound call after dialing, thus allowing the application to process the call based on the outcome. By using Call Analysis, you can determine the following:

- if the line is answered and, in many cases, how the line is answered
- if the line rings but is not answered
- if the line is busy
- if there is a problem in completing the call

GlobalCall call analysis uses 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. GlobalCall will disconnect an outbound call and report a **GCEV_CALLSTATUS** or **GCEV_DISCONNECTED** event to the application if the call is not answered within the default timeouts defined by the protocol or the **gc_MakeCall()** function. GlobalCall can also count the number of rings and report the **GCEV_CALLSTATUS** or **GCEV_DISCONNECTED** event if the maximum number of rings is reached. The maximum number of rings can be changed by using the **gc_LoadDxParm()** function to change the GlobalCall **ca_nbrdna** voice call analysis parameter, otherwise the default value of four rings is used.

2. Developing GlobalCall Analog Loop Start Applications

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

2.4.1. Call Analysis for ANAPI Protocols

Some of the country dependent protocol (*.cdp*) parameters define tone templates for recognition of call progress tones. The tone IDs created match the protocol parameter numbers (for example, parameter \$103 creates tone ID # 103).

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

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

Frequency is expressed in Hz and time duration is expressed in 10 ms units; unspecified values are set to 0. To comment out a tone template in the *.cdp* file, insert a “;” (semicolon) as the first character in all three lines of the definition. If either of the busy tones is detected, the GCEV_DISCONNECTED event is reported to the application.

When the call is in the Connected state, the following disconnection tones are available:

- for use by COs: CO_DT1 (dial tone 1), CO_BT (busy), CO_FTB (fast busy) and CO_RBT (ringback)
- for use by PBXs: PBX_BT (busy)

Only the call progress tone definitions in the *.cdp* file are used by the GlobalCall API. See the *GlobalCall Country Dependent Parameters (CDP) Reference* or your *GlobalCall Analog Protocol Release Notes* for details.

The ANAPI protocol supports call analysis via the **gc_MakeCall()** function, which uses the **flags** parameter in the ANAPI_MAKECALLBLK structure to determine if call progress and/or media type detection are enabled on a per call basis. The two flags are NO_CALL_PROGRESS and MEDIA_TYPEDETECT. The default values are such that call progress is enabled and media type detection is disabled. The bits in the **flags** parameter can be changed to enable/disable call progress and media type detection as required. If this method is used for media detection, the application must receive a GCEV_CONNECTED event before the **gc_GetCallInfo()** function can be used to get information about the type of connection. Even after the GCEV_CONNECTED event is received, the call information may not be available. Consequently, the application may need to poll for the information.

2.4.2. Call Analysis for PDKRT Protocols

The Protocol Development Kit Run-Time (PDKRT) uses default tones defined in the Dialogic Voice library for recognition of call progress tones. Any call progress tone defined by the Dialogic Voice library will be detected. See the *Call Analysis* chapter in the *Voice Software Reference - Voice Features Guide* for more information on 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

2. Developing GlobalCall Analog Loop Start Applications

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.

2.5. Header Files

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

- `anapi.h` - required when using ANAPI error codes or the `ANAPI_MAKECALLBLK` structure for call analysis.
- `gcpdkrt.h` - required when using PDK error codes, the `PDK_MAKECALL_BLK` structure for call analysis, or logging via the `gc_Start()` function.

2.6. Resource Association

For 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 GlobalCall 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.

2.7. Alarm Handling

As described in the *GlobalCall API Software Reference*, the GCEV_BLOCKED event indicates that a line is blocked and the application cannot issue call-related function calls. The GCEV_UNBLOCKED event indicates that the line has become unblocked.

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

2.8. 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 the following table.

Table 2. Reasons for Network Call Termination

| Reason/ Message | GlobalCall Result Value | ANAPI Result Value |
|-----------------------------------|------------------------------------|-------------------------------|
| Disconnect by loop current change | GCRV_NORMAL | ANRV_DISCSIG |
| Disconnect by tone | GCRV_NORMAL | ANRV_DISCTONE |

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

2.9. Run Time Configuration of the PDKRT Call Control Library

Table 3 shows the parameters of the PDKRT call control library that can be configured using the Real Time Configuration Manager (RTCM). The **gc_GetConfigData()** function can be used to retrieve the target object

2. Developing GlobalCall Analog Loop Start Applications

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 3. 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 <code>gc_GetCallInfo()</code>) | char | GC_R_O |
| Note * GC_R_O - retrieve only | | | | | |

2.10. Run Time Configuration of PDK Protocol Parameters

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 can only be changed when the channel object does not have an active call.

Protocol State Information (*.psi*) variable parameters are interpreted by the PDK run-time component (PDKRT). The names of the Protocol State Information (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 4*.

Table 4. 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 Protocol Service Layer (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. 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 names of the Protocol Service Layer (PSL) variable parameters begin with PSL_ and SYS_. The PSL parameters that can be accessed via **gc_GetConfigData()**, **gc_SetConfigData()**, and **gc_QueryConfigData()** are shown in *Table 5*.

Table 5. 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 6 shows the Set ID and Parm ID for these parameter types.

2. Developing GlobalCall Analog Loop Start Applications

Table 6. Configurable PDK Protocol Parameters

| Set ID | Parm ID | Target Object Type | Explanation | Update Flag ** |
|--|----------------------|--|--|----------------|
| PKKSET_PSI_VAR * | Dynamically assigned | GCTGT_PROTOCOL_SYSTEM, GCTGT_PROTOCOL_CHAN | Protocol state information (PSI) variable parameters. | GC_W_N |
| PKKSET_SERVICE_VAR | Dynamically assigned | GCTGT_PROTOCOL_SYSTEM, GCTGT_PROTOCOL_CHAN | Protocol service layer (PSL) variable parameter and system parameters. | GC_W_N |
| <p>Note: * indicates that CAS pattern signals and tones cannot be accessed. ** GC_W_N - update only at null state</p> | | | | |

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

The GlobalCall 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 GlobalCall application must call **gc_QueryConfigData()** to find the parameter information (set ID, parm ID, and value data type, etc.) first. For more information on these functions, refer to the *GlobalCall Application Developer's 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, GlobalCall protocol ID)
- (GCTGT_PROTOCOL_CHAN, GlobalCall line device ID)

For a given protocol, although the GCTGT_PROTOCOL_SYSTEM target object and GCTGT_PROTOCOL_CHAN target object share the same set ID and parm ID for PSI variables, they can have different values. When a new

GCTGT_PROTOCOL_CHAN target object is opened, it gets a copy of the current PSI variable configuration of GCTGT_PROTOCOL_SYSTEM target object. Under this situation, changes to the GCTGT_PROTOCOL_SYSTEM target object configuration will not affect the configuration of the GCTGT_PROTOCOL_CHAN target object. But the GCTGT_PROTOCOL_SYSTEM target object shares the same PSL variable configuration with other GCTGT_PROTOCOL_CHAN target objects.

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

NOTE: Error handling is not shown.

```
GC_PARM t_SourceParm, t_DestParm;
GC_PARM_ID t_ParmIDSt;
char t_name[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.paddress = t_name; /* Pass the PSI variable name */
memset(&t_ParmIDSt, 0, sizeof(GC_PARM_ID));
t_DestParm.pstruct = &t_ParmIDSt; /* 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_ParmIDSt.set_ID,
                      t_ParmIDSt.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);
```

2.11. Determining Protocol Version

The following sample software code demonstrates how to determine the GlobalCall protocol version you are running.

```
#include <gclib.h>
#include <gcerr.h>
#include <srllib.h>
int main()
{
    LINEDEV    ldev;
```

2. Developing GlobalCall Analog Loop Start Applications

```
GC_PARM    parm;
int        retcode;
METAEVENT  metaevent;
parm.paddress = NULL;

int mode;
#ifdef _WIN32
mode = SR_STASYNC|SR_POLLMODE;
#else
mode = SR_POLLMODE;
#endif

if (sr_setparm(SRL_DEVICE, SR_MODELTYPE, &mode) == -1)
{
    // Error processing
}
gc_Start(NULL);
retcode = gc_Open(&ldev,":P_na_an_io: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.evtttype == 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%x) -
                %s; CC(0x%x) - %s\n",
                gc_error, gc_msg, cc_error, cc_msg);
            return (gc_error);
        }
    }
}

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


3. Applying GlobalCall Functions to Analog Loop Start Applications

Certain GlobalCall functions have additional functionality or perform differently when used in an analog loop start environment. The general function descriptions in the *GlobalCall API Software Reference* do not contain detailed information on a particular technology. Detailed information in terms of the additional functionality or the difference(s) 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 *GlobalCall API Software Reference*.

The following GlobalCall analog loop start functions are described in this Chapter:

- `gc_AcceptCall()`
- `gc_AnswerCall()`
- `gc_DropCall()`
- `gc_GetANI()`
- `gc_GetCallInfo()`
- `gc_MakeCall()`
- `gc_OpenEx()`
- `gc_ReleaseCall()`
- `gc_ResetLineDev()`

See the *GlobalCall API Software Reference* for a complete listing of GlobalCall 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 GlobalCall 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.

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_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.

3.4. gc_GetANI()

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 *GlobalCall Country Dependent Parameters (CDP) Reference* for required parameter settings.

3. Applying GlobalCall Functions to Analog Loop Start Applications

3.5. 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

See the *anapi.h* header file for the equates.

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

3.6. gc_MakeCall()

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 *Function Reference* chapter of the *GlobalCall API Software Reference*. Also, see the *GlobalCall Country Dependent Parameters (CDP) Reference* that accompanies your protocol software for other timeouts that may apply to your analog protocol.

If all **timeout** values are set to 0, no timeout condition will apply.

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.

The **gc_MakeCall()** function description in the *GlobalCall API Software Reference* provides a table describing "Call Conditions and Results." In addition to the "Event/Return Value" and the "Result/Error Value" described in that table, the values described in *Table 7* apply when running Analog technology:

Table 7. Analog Call Conditions and Results

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

3. Applying GlobalCall Functions to Analog Loop Start Applications

3.7. gc_OpenEx()

The **gc_OpenEx()** function opens voice channels, voice devices or analog loop start interfaces. For 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, *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 various 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_na_an_io:V_dxxxB3C2`

GlobalCall 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.

3.8. 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.

3.9. gc_ResetLineDev()

The **gc_ResetLineDev()** function is used to ensure that voice channels are set on-hook. The **gc_ResetLineDev()** function also sets the GlobalCall 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.

3.10. 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 *Section 6.2. Debugging Applications that use PDK Protocols* for more information.

3. Applying GlobalCall Functions to Analog Loop Start Applications

3.11. 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. See *Section 3.10. gc_Start()* for more information.

For PDK protocols, the **filename** argument is ignored. The name of the log file is specified in the **CCLIB_START_STRUCT** data structure. See *Populating and Using a CCLIB_START_STRUCT* for more information.

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 voice 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.

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. For voice boards with on-board analog loop start devices, a single **gc_OpenEx()** function call opens a voice channel comprising the voice and analog loop start resources.

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 because these two resources were internally attached when opened.

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

GlobalCall™ Analog Technology User's Guide for Linux and Windows

```
1      /* Open a GlobalCall device with a voice channel and a
        analog loop start network time slot */
        if (gc_OpenEx(&linedev, ":P_na_an_io:V_dxxxB1C1", 0, &usrattr)
            == EGC_NOERR) {
            /*
             * Wait for GCEV_UNBLOCKED event.
             */
            .
            .
            .
            /* Make an outgoing call */
2      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() */
        }
        .
        .
        .
```

Legend:

- 1 The **gc_OpenEx()** function:
 - opens a GlobalCall line device using voice channel dxxxB1C1 and configures the line device to use North America Analog Protocol.
 - opens the analog loop start time slot and voice channel automaticallySCbus 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

Protocols are distributed as a separately orderable product. 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). When using PDK protocols, some parameters are dynamically updateable (that is, the parameter value can be changed while the application is running). See 2.10. *Run Time Configuration of PDK Protocol Parameters* for more information.

The *GlobalCall Country Dependent Parameters (CDP) Reference* that accompanies the protocol software lists each supported protocol and describes the modifiable parameters in the protocol's .*cdp* file.

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, Dialogic provides .*cdp* country dependent parameter files that can be modified to satisfy local requirements.

5.1. Protocol Naming Convention

The GlobalCall analog loop start protocol files use the naming convention described in the following table.

Table 8. Protocol File Naming Conventions

| Code | Description |
|--------------------------|--|
| pdk_ | Prefix for PDK protocols only. |
| cc | Two 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 | Two character technology type. Valid types are: an = analog protocol |
| fff (optional) | Defines a special software or hardware feature supported by the protocol; one to four characters. |
| d | One or two character direction indicator. Valid directions are: <ul style="list-style-type: none"> • i - inbound • o - outbound • io - inbound/outbound |

The GlobalCall analog loop start protocol component names are constructed as described in *Table 9*.

Table 9. Protocol Component Names

| Code | Description |
|--|-----------------------------------|
| <i>cc_tt_d.so</i> or <i>cc_tt_fff_d.so</i> | ANAPI protocol module for Linux |
| <i>cc_tt_d.so</i> or <i>cc_tt_fff_d.dll</i> | ANAPI protocol module for Windows |
| <i>cc_tt_d.cdp</i> or <i>cc_tt_fff_d.cdp</i> | country dependent parameter file |

| Code | Description |
|---|--|
| <i>pdk_cc_tt_d.psi</i> | PDK protocol state information file |
| <i>cc_tt_d.sdp</i> or <i>cc_tt_ffff_d.sdp</i> | PDK protocol site dependent parameter file |
| <i>cc_tt.txt</i> or <i>cc_tt_ffff.txt</i> | ANAPI protocol package release note |

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

Examples of the files included for the ANAPI North American analog loop start protocol are provided in *Table 10*.

Table 10. Sample ICAPI Protocol File Set

| Description | Protocol Files Linux | Protocol Files Windows |
|---|---------------------------------|-----------------------------------|
| Analog loop start protocol module | <i>na_an_io.so</i> | <i>na_an_io.dll</i> |
| Inbound/outbound country dependent parameters | <i>na_an_io.cdp</i> | <i>na_an_io.cdp</i> |
| Text of Release Notes | <i>na_an.txt</i> | <i>na_an.txt</i> |

For PDK protocols, we recommend that you use bi-directional protocols. Examples of the files included for the PDK North American analog protocol are provided in *Table 11*.

Table 11. Sample PDK Protocol File Set

| Description | Protocol Files Linux and Windows |
|--------------------------------|---|
| Bi-directional protocol module | <i>pdk_na_an_io.psi</i> |
| Country dependent parameters | <i>pdk_na_an_io.cdp</i> |

5.2. Protocol Components

The file types included with a protocol are:

- **protocol modules:** these files contain protocol specific information and are dynamically linked to the application as needed.
- **country dependent parameter (.cdp) file:** contains protocol related parameters. In addition to the voice parameter file loaded by the **gc_LoadDxParm()** function, GlobalCall uses a country dependent parameter (.cdp) file that defines country specific and protocol specific parameters for use by GlobalCall. For ICAPI protocols, the special parameter @0 listed at the top of the .cdp file, identifies the protocol to be run. This parameter specifies the name of the protocol module (ignoring the filename extension and without the path) to be run by the application. Two variations of the same protocol can be run if two .cdp files point to the same protocol module filename after @0. See 5.2.1. *Country Dependent Parameter (.cdp) Files* for more information.
- **site dependent parameter (.sdp) file:** this file has the same format as the .cdp file. The .sdp file is not supplied with the protocol but can be created by the user to define local information, for example, local telephone numbers as well as define switch-dependent variants and implement variants of the same basic protocol. See 5.2.2. *Site Dependent Parameter (.sdp) Files* for more information.

5.2.1. Country Dependent Parameter (.cdp) Files

Descriptions of the country dependent parameters most likely to be modified for a protocol are provided in the *GlobalCall Country Dependent Parameters (CDP) Reference* provided with the protocol software. In the following discussions, *cc* as the first 2 characters of a filename represents the 2 character country code; for example, the file *cc_an_d.cdp* would be *na_an_io.cdp* for the ANAPI North America Analog Protocol file.

Country dependent parameter (.cdp) files may be customized by modifying the file using any utility or word processor that can edit and save ASCII text.

5. Analog Protocols

Call progress tones (for example, busy tones 103 and 104 and ringback tone 105) may be useful in determining the condition of a call. To use one or more of these tones, you must ensure that one or more of the following commands are included in your `.cdp` file (or uncomment these lines) when installing your protocol:

```
*****
*   TID # 103  CO_BT   *
*****
$103 CO_BT   : 481  50 623  60
%01 cadence  :  48  10 49  10
%02 cycle    :    1
*****
*   TID # 104  CO_FBT  *
*****
$104 CO_FBT  : 479  40 619  50
%01 cadence  :  25   5 25   5
%02 cycle    :    4
*****
*   TID # 105  CO_RBT  *
*****
$105 CO_RBT  : 432  70 502  70
%01 cadence  : 187  25 419  60
%02 cycle    :    1
```

For some protocols, certain parameters must be set in the country dependent parameter (`.cdp`) file(s) to ensure proper operation of the protocol. Refer to the *GlobalCall Country Dependent Parameters (CDP) Reference* or the Release Note for your analog protocol for country dependent parameters that are most likely to be modified and for any required settings.

5.2.2. Site Dependent Parameter (.sdp) Files

PDK protocols can use site dependent parameter (`.sdp`) files. An `.sdp` file, which has the same format as a country dependent parameters (`.cdp`) file, can be created to define site-specific parameters, such as local phone numbers and switch-dependent variables. The parameter definitions in the `.sdp` file override any of the corresponding parameter definitions in the `.cdp` file, but may also include additional parameters that are supported, but not in the `.cdp` file.

By defining local parameters, such as local phone numbers and switch-dependent variables in an `.sdp` file, loss of these parameter definitions can be avoided if the protocol and or the `.cdp` files are updated.

GlobalCall™ Analog Technology User's Guide for Linux and Windows

If an *.sdp* file is being used, and a change is made to one or more parameters in the *.sdp* file, the application must be restarted before any parameter changes take effect.

NOTE: ICAPI protocols do not support *.sdp* files.

The *.sdp* file should be located **only** under the directory defined in the DLFCGPATH environment variable.

In Windows, the directory is:

x:\Program Files\Dialogic\Cfg (where **x** is a drive name)

In Linux, the directory is:

usr/dialogic/cfg

6. Debug Utilities

GlobalCall 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.

For Linux applications, the log file (*anapi.log.<pid>*, where pid = the process identification number) is generated by compiling the *ancountry.c* file with the symbol `DEBUG` defined and then setting the parameters \$11 and \$12 in the *anapi.cfg* file as indicated in *Table 12. anapi.cfg File*. To write additional information directly to the ANAPI log file, use the **anapirs_log_printf()** function. This function works like the **fprintf()** function except that a file descriptor is not used.

The debugging facilities for ANAPI and PDK protocols are described in the following sections:

- *Section 6.1. Debugging Applications that use ANAPI Protocols*
- *Section 6.2. Debugging Applications that use PDK Protocols*

6.1. Debugging Applications that use ANAPI Protocols

The *anapi.cfg* file contains a number of parameters that are useful for debugging. *Table 12* describes the debug-related parameters. Unless noted in the *GlobalCall API Software Package Release Notes*, these parameters should retain their original settings.

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

Table 12. anapi.cfg File

| Parameter | Description |
|---|--|
| \$11 | <p>Logging utility enabled (1=YES, 0=NO) : 1</p> <ul style="list-style-type: none"> • Set to 1 to enable logging, either to the screen (set \$13 parameter to 1) or to the <i>anapi.log</i> file to track all the events that occur at the device selected for monitoring (parameter \$12). • Set to 0 to ignore parameters \$12, \$13 and \$15. |
| \$12 | <p>Device handle of channel to monitor (-1=none, 0=All): 0</p> <ul style="list-style-type: none"> • A value of -1 means do not monitor any device. • A value of 0 means monitor all opened devices. |
| \$13 | <p>Echo on screen (1=YES, 0=NO): 1</p> <ul style="list-style-type: none"> • Set to 1 to send the debug information to the screen. • Set to 0 to ignore parameter. |
| \$15 | <p>Size of debug memory (1=1 event or action in memory) : 1</p> <p>The debug memory saves passed actions or events to a buffer. The built-in debug function does not use this feature. Change this parameter only if you implement your own debug function and you need a larger circular buffer than 1 event or action.</p> <ul style="list-style-type: none"> • Set to 0 to ignore feature. • Set to 1 to store one action or event in the buffer. |
| <p>NOTE: Only run the debugging and logging utilities on a limited number of channels at a time to avoid the possibility of losing events.</p> | |

6.2. Debugging Applications that use PDK Protocols

In a SpringWare environment, the GlobalCall 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.

6.2.1. Enabling and Disabling the Logging

CAUTION:

It is recommend that logging be done on an as-needed basis. Logging uses significant resources and can reduce the performance of the GlobalCall 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.

In an environment that uses SpringWare boards, 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
- 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.

Populating and Using a CCLIB_START_STRUCT

The following code show 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 13*.

Table 13. cclib_data Fields and Values

| Field Name | Field Values | Default Value |
|------------|-----------------------|--------------------------------|
| filename | Log file name | gc_pdk.log |
| loglevel | See <i>Table 14</i> . | ENABLE_FATAL or 5 |
| service | See <i>Table 15</i> . | ALL_SERVICES or 0xFFFFFFFF |
| cachedump | See <i>Table 16</i> . | WHEN_FULL or 1 |
| cachesize | Any positive integer | 1 (number of records in cache) |

6. Debug Utilities

| Field Name | Field Values | Default Value |
|-------------|-----------------------|----------------|
| channel | See <i>Table 17</i> . | 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 is an example of how to set the **cclib_data** string:

```
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"
```

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

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 tables following show the allowable values for the **loglevel**, **service**, **cachedump** and **channel** fields respectively. The values of **loglevel**, **service** and **cachedump** can be numbers (if hex format is used, the prefix 0x should be used) or symbols. 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 14 shows the valid values the log_level parameter.

Table 14. log_level Values

| log_level | 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 <i>channelimpl.cpp</i> , dx_open() returns INVALID_VOICEH. It is expected that an exception will be thrown and the log cache will be dumped to a file if possible. |
| ENABLE_WARNING | 4 | All levels above ALERT are logged. An error occurs that may make the program run abnormally. For example, in <i>channelimpl.cpp</i> , 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. For example, in <i>globalcallload.cpp</i> , if the <i>.sdp</i> file does not exist, then a log record with ALERT rather than WARNING will be issued. |
| ENABLE_INFO | 2 | All levels above DEBUG are logged. Important information that the user needs to be aware of is logged. For example, in <i>channelimpl.cpp</i> , 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. |

6. Debug Utilities

| log_level | Valid Value | Description |
|--|--------------------|---|
| ENABLE_DEBUG | 1 | All levels are logged. The lowest level with the most detailed information to help debug protocols or code step-by-step. For example, in <i>channelimpl.cpp</i> , 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 15 shows the valid values the **log_service** parameter.

Table 15. log_service Values

| log_level | 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. |

| log_level | Valid Value | Description |
|---|--------------------|--------------------------------|
| 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 (65536) | Only GCLIB service enabled. |
| Note: Values prefixed with 0x are hexadecimal values. Decimal values are shown in parenthesis. | | |

Table 16 shows the valid values for the log_cachedump parameter.

Table 16. log_cachedump Values

| log_level | 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 17 shows some examples of the log_channel parameter.

Table 17. Sample log_channel 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. |
| B1C1,20 | Channel 1 and 20 on board 1. |
| B1-4C* | All channels of board 1 to 4. |
| B1C2, B2C2,20-22 | Channel 2 of board 1, channel 2, 20, 21, and 22 on board 2. |

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 is an example of a GC_PDK_START_LOG environment variable definition:

```
set GC_PDK_START_LOG = "filename : pdktest.log; binaryfile : 1;
oglevel: ENABLE_DEBUG; service : R2MF_ENABLE | CAS_ENABLE;
cachedump : WHEN_FULL | THREAD_ON; channel : B1C1, B2C2-4;
cachesize : 10; maxfilesize : 0; mindiskfree : 20"
```

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

NOTE: The example above shows all the possible field values in the environment variable. In practice, you only need to specify the values of fields that are different than the default values.

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"
```

6.2.2. 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.

6.2.3. gc_ExtensionFunction()

For debugging purposes, the **gc_ExtensionFunction()** should only be used if requested by Dialogic Technical Support. It enables users to include debug information useful to Dialogic 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 GlobalCall call control library ID
- **linedev** is the GlobalCall 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 6.2.4. PDK_XTEN_LOG_FUNC* for more information.

6.2.4. 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 18 describes the meaning of each field in the data structure.

Table 18. PDK_XTEN_LOG_FUNC Field Descriptions

| Field | Description |
|-----------|--|
| func_no | Identifies the extension feature requested. Possible values are: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none">• 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. |

6.2.5. Extended Logging Code Example

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

```
#include <gcplib.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;
```

6. Debug Utilities

```
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

\$

\$103, 9

\$104, 9

\$11, 35

\$12, 35, 36

\$13, 35, 36

\$15, 35, 36

▪

.cdp file, 29, 32

@

@0

special parameter, 32

A

additional tones, 8

alarm handling, 12

analog loop start, 3, 11

analog loop start device, 11, 27

analog loop start devices, 11

analog signaling, 5, 11, 27

ANAPI protocols
debugging, 35

anapi.cfg, 35

anapi.h header file, 21

anapi.log file, 36

anapirs_log_printf(), 35

ANI information, 21

answering machine
detect, 5

answering machine detection, 21

application
designing and coding, 3

ASCII text, 32

audio tones, 5

B

block analog line, 6

busy, 9

busy tone, 9, 33

C

cadence break, 21

call analysis, 5, 7, 8

call disconnect, 4

call parameters, 5

call progress, 10

call progress failure tones, 9

call progress tones, 4, 5, 9, 33

call termination
network, 12

called party, 5, 6

caller ID, 20

calling party, 5, 6

cause parameter, 20

cc

country code, 32

CCLIB_START_STRUCT
using for debugging, 38

channel to monitor
\$12, 36

channel-level parameters, 3

code example
extended logging, 46

component names
protocol, 30

connection types, 21

country code, 30, 32

country dependent parameter, 29, 32

country.c, 35

D

DEBUG, 35

debug information
echo on screen, 36

debug memory
\$15, 36

debugging, 35
ANAPI protocols, 35
cclib_data fields and values, 38
enabling for PDK protocols, 37
log_cachedump values, 42
log_level values, 40
log_service values, 41
PDK protocols, 37
sample log_channel values, 43

dedicated voice resource, 27

dedicated voice resources
example, 27

delete tones, 8

destination CO, 5, 6

device monitoring, 36

devicename, 23

devicename parameter, 31

dial tone, 9

dialed digits, 6

dialing code
case-sensitive, 22

dialing mode, 22

digit detection accuracy, 6

Directory Number
DN, 20

disconnect tones, 4

disconnection
reason, 12

disconnection tone, 9

DN
Directory Number, 20

DTMF
Dual Tone Multi-Frequency, 5

DTMF digits, 6

DTMF signaling, 4, 6

Dual Tone Multi-Frequency
DTMF, 5

DX_CAP data structure, 4, 5

dx_setparm(), 4

E

echo on screen
\$13, 36

EGC_PROTOCOL, 22

enhanced call analysis, 4, 5

enhanced call analysis parameters, 4

example

dedicated voice resources, 27

extended logging, 44

code example, 46

F

fast busy, 9

fax machine

detect, 5

fax machine detection, 21

fprintf(), 35

frequency, 9

frequency overlap, 6

G

gc_AcceptCall(), 19

gc_AnswerCall(), 4, 20

gc_CallAck(), 4

gc_DropCall(), 20, 24

gc_ExtensionFunction(), 44

gc_GetANI(), 20

gc_GetCallInfo(), 20, 21

gc_GetVoiceH(), 27

gc_LoadDxParm(), 3, 4, 5, 32

gc_MakeCall(), 4, 21, 22, 28

gc_Open(), 28

gc_OpenEx(), 3, 23, 27, 31

gc_ReleaseCallEx(), 24

gc_ResetLineDev(), 24

gc_Start(), 24

gc_StartTrace(), 25

gc_WaitCall(), 4, 24

GCEV_ALERTING, 9

GCEV_DISCONNECTED, 9

GCEV_DISCONNECTED event, 12

GCEV_OFFERED event, 4

GCEV_RESETLINEDEV event, 24

GCRV_PROTOCOL, 22

GCRV_TIMEOUT, 22

Global Tone Detection, 5

GlobalCall call analysis, 5

GTD, 8

GTD tones, 7

I

inbound call, 5, 24

inbound calls, 4

info_id parameter, 21

international networks, 6

ISO

country code, 30

L

LDID

line device ID, 11

line device ID

LDID, 11

local CO, 5, 6, 29

log file, 35

logging

extended, 44

logging utility, 36
loop current, 21
loop current change, 4
loop current detection
 analog signaling, 11, 27

M

MF digits, 6
MF signaling, 6
MF tone signaling, 4
monitoring
 device, 36
multifrequency code, 6
multifrequency combinations, 6

N

naming convention
 protocol, 30
national networks, 6
network device independence, 7
network handle, 27
number of rings, 4
numberstr parameter
 dialing string, 22

O

options
 protocol, 29
outbound call, 4, 5
 progress, 8

P

PBX, 9

PDK protocol
 file set for ICAPI, 31
PDK protocols
 debugging, 37
 enabling debugging, 37
PDK_XTEN_LOG_FUNC, 45
PerfectCall, 5
pre-existing tones, 7
process identification number, 35
protocol
 component names, 30
 file set for PDK, 31
 naming convention, 30
 service layer parameters, 14
 troubleshooting, 35
protocol disk, 32
protocol module, 32
 ICAPI, 13
 PDK, 13
protocol parameter numbers, 9
protocols, 29
pulse dialing, 4, 6

R

remote end, 20
resource sharing, 11
ring detection
 analog signaling, 11, 27
ringback, 9
ringback tone, 6, 9, 33
rings parameter, 4, 19, 20
rotary dialing, 6

S

SCbus, 28

Service layer parameters, 14

signaling frequencies, 6

signaling information, 5

SIT, 8

 Special Information Tones, 8

Special Information Tones

 SIT, 8

special parameter @0, 32

T

telephone number
 called party, 5

timeout, 21, 22

tone definition, 10

tone definitions, 7

tone ID, 8, 9

tone resource, 27

tone template, 9

tone templates, 9

tones downloaded, 8

troubleshooting, 35

U

unblock analog line, 6

V

voice channel, 7

voice channel parameter (.vcp)
 ASCII text file, 3

voice detection, 21

voice device, 27

voice handle, 27

voice resource, 27
 dedicated, 27

voice resources
 dedicated, 27

