IP Gateway (Global Call) Object Oriented

Demo Guide

October 2002

05-1825-001

Publication Date: October 2002

Document Number: 05-1825-001

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 Communications Systems Products website at:
http://www.intel.com/network/csp/

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

Contents

Figures

Tables

About This Publication

This section describes the purpose of the guide, the intended audience, and references to other documents that may be useful to the user.

Purpose

This guide provides information on the IP Gateway (Global Call) Object Oriented demo that is available with your Intel® Dialogic® system release. This guide describes the demo, its requirements, and details on how it works.

Intended Audience

This guide is intended for application developers who will be developing a PSTN-IP gateway application using the Global Call API. Developers should be familiar with the C++ programming language and either the Windows* or Linux* programming environments.

This information is intended for:

- Distributors
- Toolkit Developers
- Independent Software Vendors (ISVs)
- Value Added Resellers (VARs)
- Original Equipment Manufacturers (OEMs)

How to Use This Publication

Refer to this publication after you have installed the hardware and the system software.

This publication assumes that you are familiar with the Windows or Linux operating system and the C++ programming language.

The information in this guide is organized as follows:

- Chapter 1, “Demo Description” introduces you to the demo and its features

- Chapter 2, "System Requirements" outlines the hardware and software required to run the demo
- Chapter 3, "Preparing to Run the Demo" describes the preparations required before running the demo
- Chapter 4, "Running the Demo" describes how to run the demo
- Chapter 5, "Demo Details" provides details on how the demo works
- Chapter 6, "Demo State Machines" provides details on the demo state machines

Related Information

See the following for more information:

- *System Release 6.0 Release Update* for information on problems fixed, known problems and workarounds, compatibility issues and last minute updates not documented in the published information.
- *DM3 for Linux Configuration Guide*
- *Intel® NetStructure™ on DM3 Architecture for cPCI on Windows Configuration Guide*
- *Intel® NetStructure™ IPT Series for Linux Configuration Guide*
- *Intel® NetStructure™ IPT Series on Windows Configuration Guide*
- *http://developer.intel.com/design/telecom/support/* (for technical support)
- *http://www.intel.com/network/csp/* (for product information)

Demo Description 1

The IP Gateway (Global Call) Object Oriented demo is an object-oriented host-based application that demonstrates using the Global Call API to build a PSTN-IP gateway. The demo source code can be used as sample code for those who want to begin developing an application from a working application. The demo is not designed to implement a complete gateway and it lacks features such as least-cost routing, etc.

The IP Gateway (Global Call) Object Oriented demo supports the following features:

- Accepts IP calls
- Places IP calls
- Accepts PSTN calls
- Places PSTN calls
- Configuration file
- Command line options
- Output log files
- Printing to the monitor
- QoS

The IP Gateway (Global Call) Object Oriented demo is a cross-OS demo, running under the Windows or Linux environments. Most of the differences in the environments are handled directly by the programming interface and are transparent to the user. Other differences, due to inherent differences in the operating systems, are handled by the Platform Dependency Library (PDL). For more information about the PDL refer to the source code in the *pdl_win* or *pdl_linux* directories.

System Requirements 2

This chapter discusses the system requirements for running the IP Gateway (Global Call) Object Oriented demo. It contains the following topics:.

2.1 Hardware Requirements

To run the IP Gateway (Global Call) Object Oriented demo, you need:

- One of the following:
  - Intel® NetStructure™ DM/IP Series board
  - Intel® NetStructure™ IPT Series board
    - also requires an Intel® NetStructure™ DM/V-A series board for PSTN connection
- IP network cable

For other hardware requirements, such as memory requirements, see the *Release Guide* for the system release you are using.

2.2 Software Requirements

To run the IP Gateway (Global Call) Object Oriented demo, you need the Intel® Dialogic® System Software 6.0 for Linux or Windows. For a list of operating system requirements see the *Release Guide* for the system release you are using.

See Chapter 3, “Compiling and Linking” for a list of compilers that may be used with this demo. Using a non-supported compiler may cause unforeseen problems in running the demo.

Preparing to Run the Demo 3

This chapter discusses the preparations necessary to run the IP Gateway (Global Call) Object Oriented demo. It provides information about the following topics:

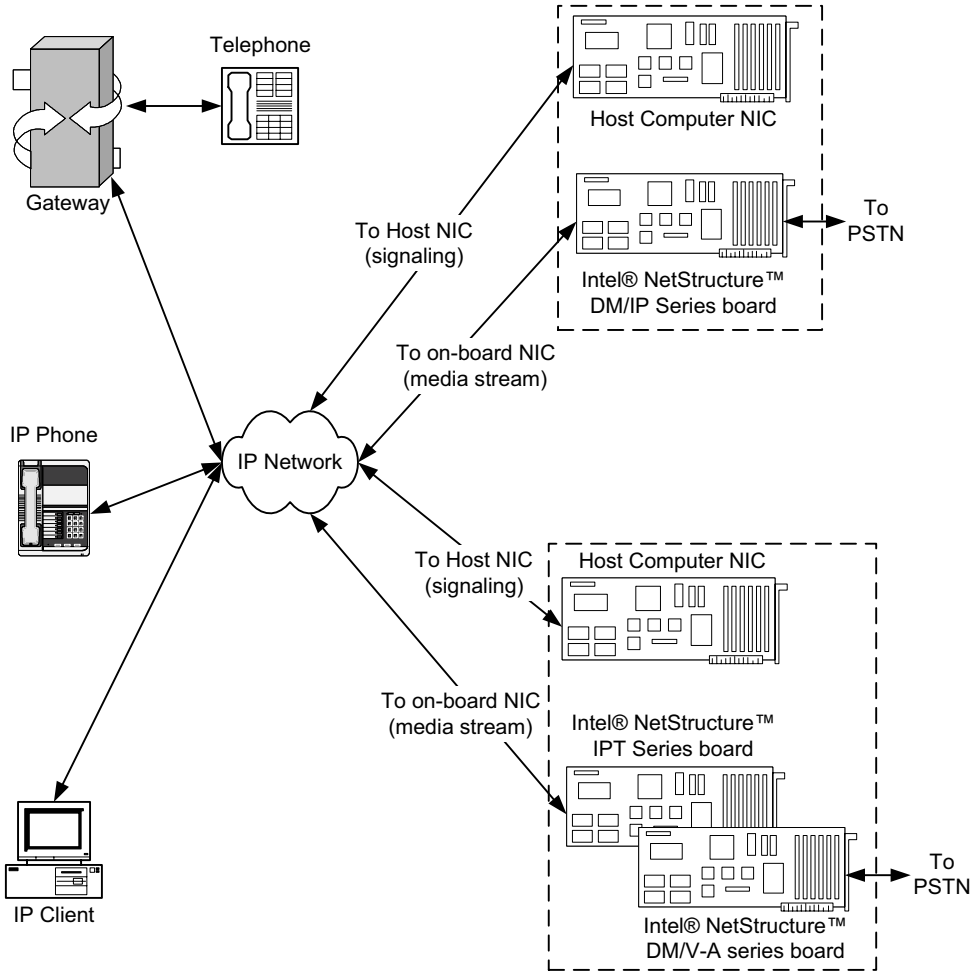
3.1 Connecting to External Equipment

There are two possible hardware configurations for the IP Gateway (Global Call) Object Oriented demo:

- Using the NIC and PSTN connection on the front end, as well as the host NIC for signaling - used with Intel® NetStructure™ DM/IP Series boards
- Using the on-board NIC and a PSTN interface board, as well as the host NIC for signaling - used with Intel® NetStructure™ IPT Series board

Figure 1 illustrates each of the possible configurations.

Figure 1. Connecting to External Equipment



3.2 Editing Configuration Files

This section discusses how to configure the demo for your system. It contains the following topics:

- File Location
- Editing the gateway_r4.cfg Configuration File

3.2.1 File Location

Before running the IP Gateway (Global Call) Object Oriented demo, modify the *gateway_r4.cfg* file to reflect your system environment. Use a text editor and open the file from:

- Windows: *C:\Program Files\dialogic\demos\gc_demos\ipdemos\gateway_r4_ood\release*

- Linux: */usr/dialogic/demos/ipt/gc_demos/gateway_r4_ood/*

3.2.2 Editing the gateway_r4.cfg Configuration File

Below is an example of the *gateway_r4.cfg* file. Update the following information:

Source

Source (originator) address

The following prefixes must be used when describing a source or destination address:

Prefix	Type	Description
NAME:	H323-ID	H323-ID is an “alias address,” such as “Intel Corp.,” usually resolved by a gatekeeper. The H323-ID consists of a string of ISO/IEC 10646-1 characters as defined in H.225.0. It may be a user name, conference name, email name, or other identifier.
TEL:	e164	A unique numeric phone number
TA:	Target Address	The IP address of the remote gateway

Destination

IP address of the NIC on the destination GW host. See Source (originator) address for a description of the prefixes that must be used.

RemotePhoneNumber

Destination phone number to call. It is transferred during call establishment to target gateway.

LocalPhoneNumber

The number used for PSTN calls

pstnProtocol

The PSTN protocol supported by the gateway. Possible values are: T1, E1, ISDN (including NFAS), CAS.

DTMFmode

Specifies how DTMF tones are transmitted. Possible values are: RTPInBand (usually used with G.711 coders), OutOfBand (usually used with low bandwidth coders, e.g., GSM), RTPRFC2833.

ipProtocol

The IP Protocol used for opening the IP line devices. Values are: H323, SIP, both.

AudioRxCodecs

Describes the receive voice coder. The parameters are as follows:

- CoderType – The type of coder. See the *System Release Update* for specific information about coder support in this release.
- CoderFramesPerPkt – Specify the number of frames per packet for the selected coder. See the *System Release Update* for specific information about coder support in this release.
- CoderVAD – Specify if VAD is active. See the *System Release Update* for specific information about coder support in this release.

AudioTxCodecs

Describes the transmit voice coder. See AudioRxCodecs for a description of the parameters.

Data Codecs

Describes the fax coder parameters. See AudioRxCodecs for a description of the parameters.

Quality of Service

The application can set threshold values to monitor the quality of service during calls. A fault occurs when the result of a measurement of a QoS parameter crossed a predefined threshold. A success occurs when the result of a measurement of a QoS parameter did not cross a predefined threshold. The QoS parameters are measured during time intervals, starting when a call is established. The following parameters are supported:

- MediaAlarmLostPackets – indicates that the percentage of packets lost during a call exceeded its threshold value
- MediaAlarmJitter – indicates that the jitter (as defined in RFC 1889) exceeded its threshold value

QoS Attributes

Each parameter has six attributes:

- Threshold – defines when a QoS parameter is in a fault condition. A fault occurs when the result of a measurement of a QoS parameter crossed the Threshold value.
- DebounceOn – the time during which faults are measured (in msec., must be multiple of Interval)
- DebounceOff – the time during which successes are measured (in msec., must be multiple of Interval)
- Interval – the amount of time between two QoS parameter measurements (in multiples of 100 msec)
- Percent_Fail – the threshold of failures during the DebounceOn time (expressed as a percentage of failures)
- Percent_Success – the threshold of successes during the DebounceOn time (expressed as a percentage of successes)

The default values are as follows:

	Threshold	DebounceOn	DebounceOff	Interval	Percent_ Fail	Percent_ Success
Lost packets	20	10000	10000	1000	60	40
Jitter	60	20000	60000	5000	60	40

Display

Display information passed to destination gateway during call establishment

IPT_UUI

User to User Information string. The information is sent before the Connected state.

UII

User Input Indication string to send. The maximum string length is 256 characters (MAX_STRING).

NonStdParm

Non-standard parameter data to send

NonStdCmd

Non-standard command string to send. The maximum string length is 256 characters (MAX_STRING).

ObjId
Object ID

Q931Facility
Facility data to send on the Q.931 channel. The maximum string length is 256 characters (MAX_STRING).

Sample Configuration File

```
Channel = 1-120
{
   Source = NAME:Dialogic Corp.
   Destination = TA:10.242.214.25
   RemotePhoneNumber = 23
   LocalPhoneNumber = 26
   pstnProtocol = isdn
   DTMFmode = inBand
   ipProtocol = H323

   AudioRxCodecs
   {
CoderType = g711mulaw
CoderFramesPerPkt = 30
CoderVAD = 0
   }

      AudioTxCodecs

   {
CoderType = g711mulaw
CoderFramesPerPkt = 30
CoderVAD = 0
   }

   DataCodecs
   {
CoderType = t38
   }

   MediaAlarmLostPackets
   {
       Threshold       = 20    # Threshold value
       DebounceOn      = 10000 # Threshold debounce ON
       DebounceOff     = 10000 # Threshold debounce OFF
       Interval        = 1000  # Threshold Time Interval (ms)
       PercentSuccess  = 60    # Threshold Success Percent
       PercentFail     = 40    # Threshold Fail Percent
   }

   MediaAlarmJitter
   {
       Threshold           = 60    # Threshold value
       DebounceOn          = 20000 # Threshold debounce ON
       DebounceOff         = 60000 # Threshold debounce OFF
       Interval            = 5000  # Threshold Time Interval (ms)
       PercentSuccess      = 60    # Threshold Success Percent
       PercentFail         = 40    # Threshold Fail Percent
   }
```

```
   Display = GATEWAY_Chan1
   IPT_UUI = User_to_User_1
   UII = 12345
   NonStdParm = NSP_Chan1
   NonStdCmd = NSC_Chan1
   ObjId = 2 16 840 1 113741
   Q931Facility = facility 01
}
```

3.3 Compiling and Linking

Compile the project within the following environments:

- Windows
  - Visual C++ environment, version 6
- Linux
  - g++

If you have added or changed files, to compile the project put the files in *dialogic\samples\gc_demos\ipdemos\gateway_r4_ood*.

Set gateway_r4_ood as the active project and build in debug mode.

Running the Demo 4

This chapter discusses how to run the IP Gateway (Global Call) Object Oriented demo. It contains the following topics:

4.1 Starting the Demo

Windows

Select Run from the Start Menu. The demo executable file can be found in: *C:\Program Files\dialogic\demos\gc_demos\ipdemos\gateway_r4_ood\release\gateway_r4_ood.exe*. Click OK to run the IP Gateway (Global Call) Object Oriented demo using the default settings.

Linux

The demo executable file can be found in: */usr/dialogic/demos/ipt/gc_demos/gateway_r4_ood/gateway_r4_ood*.

4.2 Demo Options

To specify certain options at run-time, launch the demo from a command line, using any of the switches listed in Table 1.

Table 1. Command Line Switches

Switch	Action	Default
-c <filename>	Configuration file name	-c *gateway_r4.cfg*
-d<*n*>	Sets Debug Level (0-4): • 0-FATAL – used when one or more channels are deadlocked. • 1-ERROR – used when the application receives a failure which doesn't cause the channel to be deadlocked. • 2-WARNING – used when some problem or failure occurred without affecting the channel's usual action. • 3-TRACE – used at the start of the application entrance or the start of any function. • 4-INFO – prints data related to a specific action. **Note:** Debug level is inclusive; higher levels include all lower levels	-d0 (Fatal)
-h or ?	Prints the command syntax to the screen	Off
-l<*n,…*>	Printouts will be printed into channel log files. If 'all' follows the –l, log files will be created for all available channels. If a list of channels in the following format: C1-C2, C3-C4, C5 follows the –l, log files are created for the channel ranges or specific channels specified in the list. If the "–l" option is not used, prints go to the stdout, for the first 2 channels only (to keep from overloading the CPU, and more convenient for viewing printouts).	Disabled
-m<*n,…*>	Enables printing channel information to the monitor, in addition to printing to the log file. A maximum of 2 channels may be printed.	Disabled
-n<*n*>	Sets the number of gateway channels	The lesser of Voice Devices or IP devices
-q	Activates Quality of Service	Disabled

4.3 Using the Demo

The demo always waits for input from the keyboard. While the demo is running, you may enter any of the commands listed in Table 2:

Table 2. Runtime Keyboard Commands

Command	Function
c or C	Prints channel statistics to file (statistics.log)
d<n> or D<n>	Change debug level during runtime, where <n> is the debug level
f or F followed by <channel number>	Send Q.931 facility message from the .cfg file
m or M followed by <channel number>	Print log files for up to 2 channels to the screen
n or N followed by <channel number>	Send H.245 non-standard command from the .cfg file
q or Q or Ctrl+c	Terminates the application
u or U followed by <channel number>	Send H.245 User Input Indication message from the .cfg file

4.4 Stopping the Demo

The IP Gateway (Global Call) Object Oriented demo runs until it is terminated. Press “q” or “Q” or “Ctrl+c” to terminate the demo application.

Demo Details 5

This chapter discusses the IP Gateway (Global Call) Object Oriented demo in more detail. It contains the following topics:

5.1 Files Used by the Demo

This section lists the files used by the demo. It contains the following information:

- Demo Source Code Files
- Utility Files
- PDL Files

5.1.1 Demo Source Code Files

In Windows the source code files listed in Table 3 are located in:
C:\Program Files\dialogic\demos\gc_demos\ipdemos\gateway_r4_ood.

In Linux the source code files listed in Table 3 are located in:
/usr/dialogic/demos/ipt/gc_demos/gateway_r4_ood/.

Table 3. Source Files Used by the IP Gateway (Global Call) Object Oriented Demo

Directory	File Name	Purpose
gateway_r4_ood	channel.cpp	Implements the operations of the Channel class
gateway_r4_ood	channel.h	Function prototype for channel.cpp
gateway_r4_ood	configuration.cpp	Implements the operations of the Configuration class
gateway_r4_ood	configuration.h	Function prototype for configuration.cpp
gateway_r4_ood	defs.h	Global definitions
gateway_r4_ood	gateway_r4_ood.ver	Demo version information
gateway_r4_ood	gccallcontrol.cpp	Implements the operations of the GCCallControl class
gateway_r4_ood	gccallcontrol.h	Function prototype for gccallcontrol.cpp
gateway_r4_ood	gcdevice.cpp	Implements the operations of the GCDevice class
gateway_r4_ood	gcdevice.h	Function prototype for gcdevice.cpp

Table 3. Source Files Used by the IP Gateway (Global Call) Object Oriented Demo (Continued)

Directory	File Name	Purpose
gateway_r4_ood	gwcall.cpp	Implements the operations of the GWCall class
gateway_r4_ood	gwcall.h	Function prototype for gwcall.cpp
gateway_r4_ood	incfile.h	Function prototype for Global Call and R4 functions
gateway_r4_ood	ipboard.cpp	Implements the operations of the IPBoard class
gateway_r4_ood	ipboard.h	Function prototype for ipboard.cpp
gateway_r4_ood	ipcallcontrol.cpp	Implements the operations of the IPCallControl class
gateway_r4_ood	ipcallcontrol.h	Function prototype for ipcallcontrol.cpp
gateway_r4_ood	ipdevice.cpp	Implements the operations of the IPDevice class
gateway_r4_ood	ipdevice.h	Function prototype for ipdevice.cpp
gateway_r4_ood	main.cpp	Contains the main function and the Wait for Key
gateway_r4_ood	main.h	Function prototype for main.cpp
gateway_r4_ood	pstnboard.cpp	Implements the operations of the DigitalPstnBoard class
gateway_r4_ood	pstnboard.h	Function prototype for digitalpstnboard.cpp
gateway_r4_ood	pstncallcontrol.cpp	Implements the operations of the PstnCallControl class
gateway_r4_ood	pstncallcontrol.h	Function prototype for pstncallcontrol.cpp
gateway_r4_ood	pstndevice.cpp	Implements the operations of the DigitalPstnDevice class
gateway_r4_ood	pstndevice.h	Function prototype for digitalpstndevice.cpp
gateway_r4_ood	r4board.cpp	Implements the operations of the R4Board class
gateway_r4_ood	r4board.h	Function prototype for r4logicalboard.cpp
gateway_r4_ood	r4device.cpp	Implements the operations of the R4Device class
gateway_r4_ood	r4device.h	Function prototype for r4device.cpp
gateway_r4_ood	resourcemanager.cpp	Implements the operations of the ResourceManager class
gateway_r4_ood	resourcemanager.h	Function prototype for resourcemanager.cpp
gateway_r4_ood (Linux only)	gateway_r4.cfg	Demo configuration file
gateway_r4_ood (Linux only)	makefile	Linux compilation file
gateway_r4_ood (Linux only)	gateway_r4_ood	Linux executable
gateway_r4_ood (Windows only)	gateway_r4_ood.dsp	Visual C++ project file
gateway_r4_ood (Windows only)	gateway_r4_ood.dsw	Visual C++ project workspace
gateway_r4_ood (Windows only)	gateway_r4_ood.rc	Resource file
gateway_r4_ood (Windows only)	resource.h	Microsoft Developer Studio generated include file used by gateway_r4_ood.rc

Table 3. Source Files Used by the IP Gateway (Global Call) Object Oriented Demo (Continued)

Directory	File Name	Purpose
gateway_r4_ood/release (Windows only)	gateway_r4.cfg	Demo configuration file
gateway_r4_ood/release (Windows only)	gateway_r4_ood.exe	Demo executable

5.1.2 Utility Files

In Windows the utility files listed in Table 4 are located in:
C:\Program Files\dialogic\demos\gc_demos\ipdemos\utilcpp.

In Linux the utility files listed in Table 4 are located in:
/usr/dialogic/demos/ipt/gc_demos/utilcpp/.

Table 4. Utility Files Used by the IP Gateway (Global Call) Object Oriented Demo

Directory	File Name	Purpose
utilcpp	utilcpp.ver	Utility library version information
utilcpp	log.cpp	Debugging functions
utilcpp	log.h	Function prototype for libdbg.c
utilcpp (Windows only)	utilcpp.dsw	Utility library Visual C++ workspace
utilcpp (Windows only)	utilcpp.dsp	Utility library Visual C++ project file
utilcpp\release (Windows only)	utilcpp.lib	Compiled Utility library
/utilcpp (Linux only)	makefile.utilcpp	Compilation file
/utilcpp (Linux only)	libutilcpp.a	Compiled Utility library

5.1.3 PDL Files

In Windows the PDL files listed in Table 5 are located in:
C:\Program Files\dialogic\demos\gc_demos\ipdemos\pdl_win.

In Linux the PDL files listed in Table 6 are located in:
/usr/dialogic/demos/ipt/gc_demos/utilcpp/pdl_linux/.

Table 5. PDL Files Used by the IP Gateway (Global Call) Object Oriented Demo - Windows OS

Directory	File Name	Purpose
pdl_win	iptransport.cpp	PDL IP transport functions
pdl_win	iptransport.h	Function prototype for iptransport.cpp
pdl_win	pdl.c	Platform dependency functions
pdl_win	pdl.h	Function prototype for pdl.c
pdl_win	pdl.ver	PDL version information
pdl_win	pdl_win.dsp	PDL Visual C project file
pdl_win	pdl_win.dsw	PDL Visual C workspace
pdl_win\release	pdl_win.lib	Compiled PDL library

Table 6. PDL Files Used by the IP Gateway (Global Call) Object Oriented Demo - Linux OS

Directory	File Name	Purpose
pdl_linux	iptransport.cpp	PDL IP transport functions
pdl_linux	iptransport.h	Function prototype for iptransport.cpp
pdl_linux	libpdl.a	Compiled PDL library
pdl_linux	makefile.pdl	Compilation file
pdl_linux	pdl.c	Platform dependency functions
pdl_linux	pdl.h	Function prototype for pdl.c
pdl_linux	pdl.ver	PDL version information

5.2 Programming Model Classes

This section presents basic information about the IP Gateway (Global Call) Object Oriented demo classes. It contains the following information:

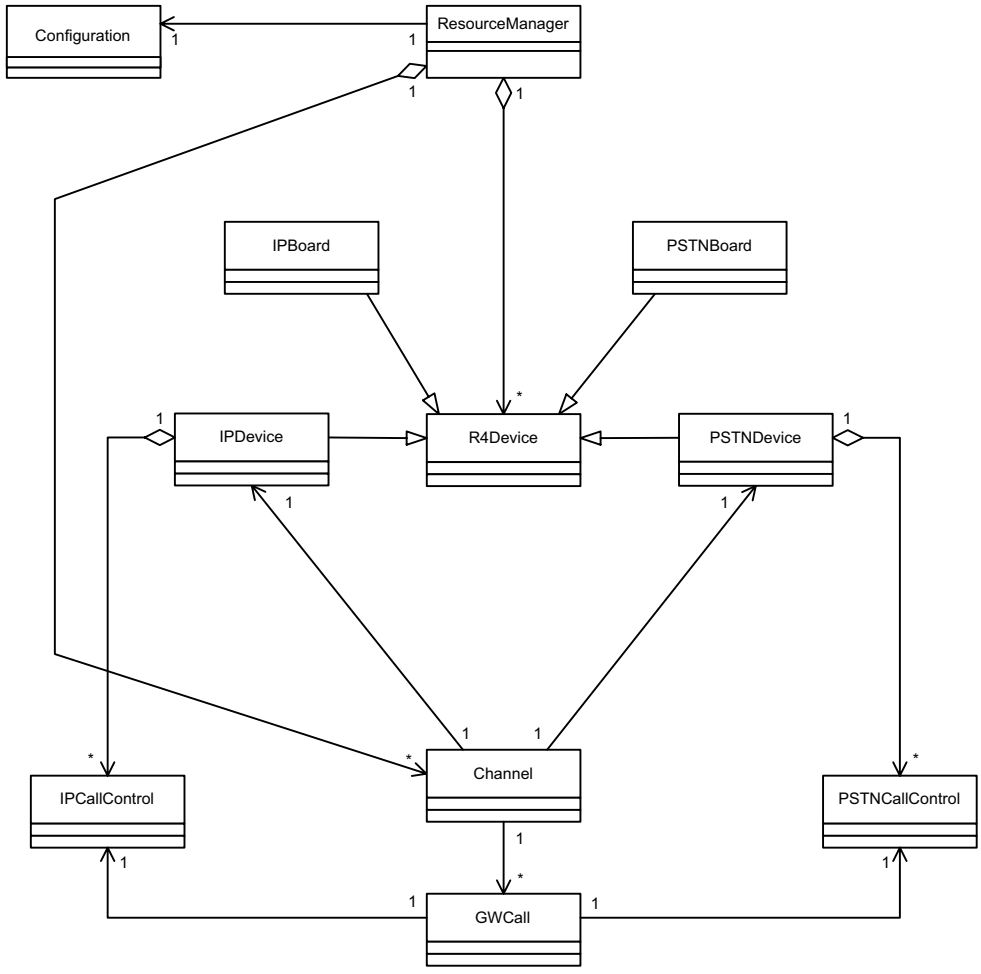
- Class Diagram
- ResourceManager Class
- IPBoard Class
- R4Board Class
- PSTNBoard Class
- IPDevice Class
- R4Device Class
- PSTNDevice Class
- GCDevice Class
- GCCallControl Class
- PSTNCallControl Class

- IPCallControl Class
- Configuration Class
- Channel Class
- GWCall Class

5.2.1 Class Diagram

The following class diagram describes the relationship among the classes.

Figure 2. IP Gateway (Global Call) Object Oriented Class Diagram



5.2.2 ResourceManager Class

The ResourceManager class' main role is to initialize the R4 resources. It handles the R4 mechanism and reflects the resource status. It contains the following data:

- all system channels
- configuration object for initialization
- maps R4 device handles to channels
- all detected R4 boards

The ResourceManager class attributes are described in Table 7.

Table 7. ResourceManager Class Attributes

Name	Access Privilege	Type	Description
m_devicesQueue	private	R4DeviceQueue [MAX_DEVICE_TYPE]	Array of device pointer queues. The array index is defined as board type (IP, PSTN). It includes all the devices in the system.
m_channelsQueue	private	ChannelQueue	A queue of all the channels created in the system.
configuration	private	Configuration	An instance of the Configuration class used to determine the configuration of the system during initialization.
m_channelNum	private	int	The number of channels that the demo will work with (this number is the minimum of devices of each type and the user requested -n option).
m_printToMonitor	private	Int[MAX_PRINT_TO_MONITOR]	The channels that are printing to the monitor
m_numOfPSTNBoards	private	int	The number of PSTN boards detected in the system.
m_numOfIPBoards	private	int	The number of IP boards detected in the system.
m_initLog	private	static Log	A log instance used during initialization. All the printouts are sent to the monitor, after which it is destructed. This instance is necessary because during the initialization, there are no channel instances and therefore no log instances. In order to see logs during initialization, the application creates a global log instance for all the devices during initialization and kills it after creating the channel objects and attributing the devices to channels.

5.2.3 IPBoard Class

The IPBoard class' main role is to manage the IP device database. It contains all the IP devices available in the system and reflects the IP device repository.

The IPBoard inherits the R4Board Class attributes. The IPBoard class attributes are described in Table 8.

Table 8. IPMediaBoard Class Attributes

Name	Access Privilege	Type	Description
m_rasInfo	private	RASInfo	Includes the information for RAS, read from the configuration file during initialization

5.2.4 R4Board Class

The R4Board class' main role is to provide all common functionality for all R4 logical boards. It opens the boards and gets all the information about the devices. The R4BoardClass is the base class for all R4 logical boards containing the common attributes and operations. It represents any R4 logical board.

The R4Board class attributes are described in Table 9.

Table 9. R4Board Class Attributes

Name	Access Privilege	Type	Description
m_boardNumber	protected	int	The board number - used in setting the device names found on the board
m_numOfDevices	protected	int	Number of devices available on the board

5.2.5 PSTNBoard Class

The PSTNBoard class' main role is to provide all common functionality of all R4 PSTN boards. The PSTNBoard class is the base class for the PSTN boards, containing the common attributes and operations.

The PSTNBoard class inherits all its attributes from the R4Board Class.

5.2.6 IPDevice Class

The IPDevice class' main role is to provide Global Call functionality for IP devices. It represents the IP devices and reflects the device status and manages all calls related to it.

The IPDevice inherits the GCDevice class attributes. The IPDevice class attributes are described in Table 10.

Table 10. IPDevice Class Attributes

Name	Access Privilege	Type	Description
m_callControls	private	IPCallControl{MAX_CALL_CONTROLS]	Contains the call control objects connected to this device

5.2.7 R4Device Class

The R4Device class' main role is to provide all common functionality for all R4 devices. It is the base class for all R4 line devices that can be opened using **gc_OpenEx( )**. It contains all the common attributes and operations for all R4 devices.

The R4Device class attributes are described in Table 11.

Table 11. R4Device Class Attributes

Name	Access Privilege	Type	Description
m_name	protected	char	The device name, e.g. ipmB1C1
m_inService	protecte	bool	True when the device is available
m_lineDevice	protected	LINEDEV	The device handle (valid after opening)
m_pLog	protected	Log*	The device log instance

5.2.8 PSTNDevice Class

The PSTNDevice class' main role is to provide all common functionality of all R4 devices. It is the base class for PSTN R4 devices.

The PSTNDevice inherits the GCDevice Class attributes. The PSTNDeviceclass attributes are described in Table 12.

Table 12. PSTNDevice Class Attributes

Name	Access Privilege	Type	Description
m_callControls	private	PSTNCallControl[MAX_CALL_CONTROLS]	Contains the call control objects connected to this device

5.2.9 GCDevice Class

The GCDevice class' main role is to provide all common functionality of all R4 signaling devices. It is the base class for R4 signaling devices.

The GCDevice inherits the R4Device Class attributes. The GCDeviceclass attributes are described in Table 13.

Table 13. GCDevice Class Attributes

Name	Access Privilege	Type	Description
m_pChannel	protected	Channel*	Points to the channel containing this device
m_txTimeSlot	protected	unsigned int	The transmit time slot of the device

5.2.10 GCCallControl Class

The GCCallControl class' main role is to provide all common functionality of GC call control devices. It is the base class for all GC call control devices and contains the common attributes and operations.

The GCCallControl class attributes are described in Table 14.

Table 14. GCCallControl Class Attributes

Name	Access Privilege	Type	Description
m_crn	protected	CRN	The call reference number of the PSTN call control object
m_pLog	protected	Log*	The log object used to control the CC object printouts
m_stateFunctionArray	protected	StateFunction[MAX_STATE]	State machine function array
m_lineDevice	protected	LINEDEV	The line device of the device that the call control belongs to
m_currentState	protected	E_StateMachine	The current state in the state machine
m_dnisEnabled	public	static bool	True if DNIS is used

5.2.11 PSTNCallControl Class

The PSTNCallControl class' main role is to provide a Global Call functionality interface to manage a call. It reflects the PSTN call status to the other classes.

The PSTNCallControl class inherits the GCCallControl Class attributes. The PSTNCallControl attributes are described in Table 15.

Table 15. PSTNCallControl Class Attributes

Name	Access Privilege	Type	Description
m_callInfo	private	PSTNCallInfo	Contains information connected to the PSTN call

5.2.12 IPCallControl Class

The IPCallControl class' main role is to provide the IP protocol functionality interface. It controls one IP call and reflects the call status to the other classes.

The IPCallControl class inherits the GCCallControl Class attributes. The IPCallControl attributes are described in Table 16.

Table 16. IPCallControl Class Attributes

Name	Access Privilege	Type	Description
m_callInfo	private	IPCallInfo	Information connected to the IP call

5.2.13 Configuration Class

The Configuration class' main role is to provide an interface to get the needed configuration data. It contains all the needed data structures to parse and save the system configuration (the configuration file and the command line options) and reflects the system configuration to the other classes.

The Configuration class attributes are described in Table 17.

Table 17. Configuration Class Attributes

Name	Access Privilege	Type	Description
m_userChannels	private	unsigned int	Indicates the number of channels that the demo will work with.
m_chanInfo	private	ChannelInfoQueue	Queue that contains all the channel information read from the configuration file, such as Tx coder information, the print to log file flag, and the phone number to call.
m_boardInfo	private	BoardInfoQueue	Queue that contains all the board information read from the configuration file.
m_cfgFile	private	char*	The configuration file name
m_alarmQoSFile	private	FILE*	Points to the QoS file
m_QoSFile	private	char*	The name of the QoS log file

Table 17. Configuration Class Attributes (Continued)

Name	Access Privilege	Type	Description
m_logLevel	private	E_LogLevel	The log level from the command line
m_stage	private	unsigned char	The stage of parsing the configuration file
m_line	private	int	The line in the configuration file currently being parsed
m_firstSession	private	long	Used to fill the channel information from the configuration file
m_lastSession	private	long	Used to fill the channel information from the configuration file

5.2.14 Channel Class

The Channel class' main role is to control all devices related to a call. It contains one IP device, one PSTN device and all calls related to these devices.

The Channel class attributes are described in Table 18.

Table 18. Channel Class Attributes

Name	Access Privilege	Type	Description
m_pIPMediaDevice	private	IPDevice*	The channel IP device
m_pPSTNDevice	private	IPSTNDevice*	The channel PSTN device
m_channelInfo	private	ChannelInfo	Configuration information about the channel
m_pLog	private	Log*	The channel log object
m_staticsInfo	private	ChannelStatistics	Channel statistical information
m_channelId	private	unsigned int	The channel identifier
m_GWCalls	private	GWCALL_LIST	List of the calls initiated on the channel

5.2.15 GWCall Class

The GWCall class' main role is to control all resources related to a call. It contains all the resources needed to establish a call. The GWCall class reflects the intersection of call resource status.

The GWCall class attributes are described in Table 19.

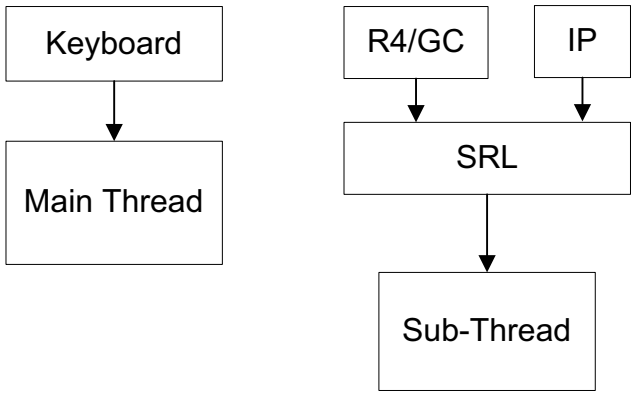
Table 19. GWCall Class Attributes

Name	Access Privilege	Type	Description
m_pPSTNCallControl	public	PSTNCallControl	PSTN call control object of the call, handling the PSTN call control
m_pIPCallControl	public	IPCallControl*	IP call control object of the call, handling the IP call control
m_currentState	private	E_StateMachine	Current channel state
m_stateFunctionArray[MAX_STATE]	private	int(stateFunction*)(const long eventType, METAEVENT* eventData)	Array of state machine functions
m_pLog	public	Log*	Log object of the call

5.3 Threads

The IP Gateway (Global Call) Object Oriented demo operates with two threads, as shown in Figure 3.

Figure 3. IP Gateway (Global Call) Object Oriented Demo Threads



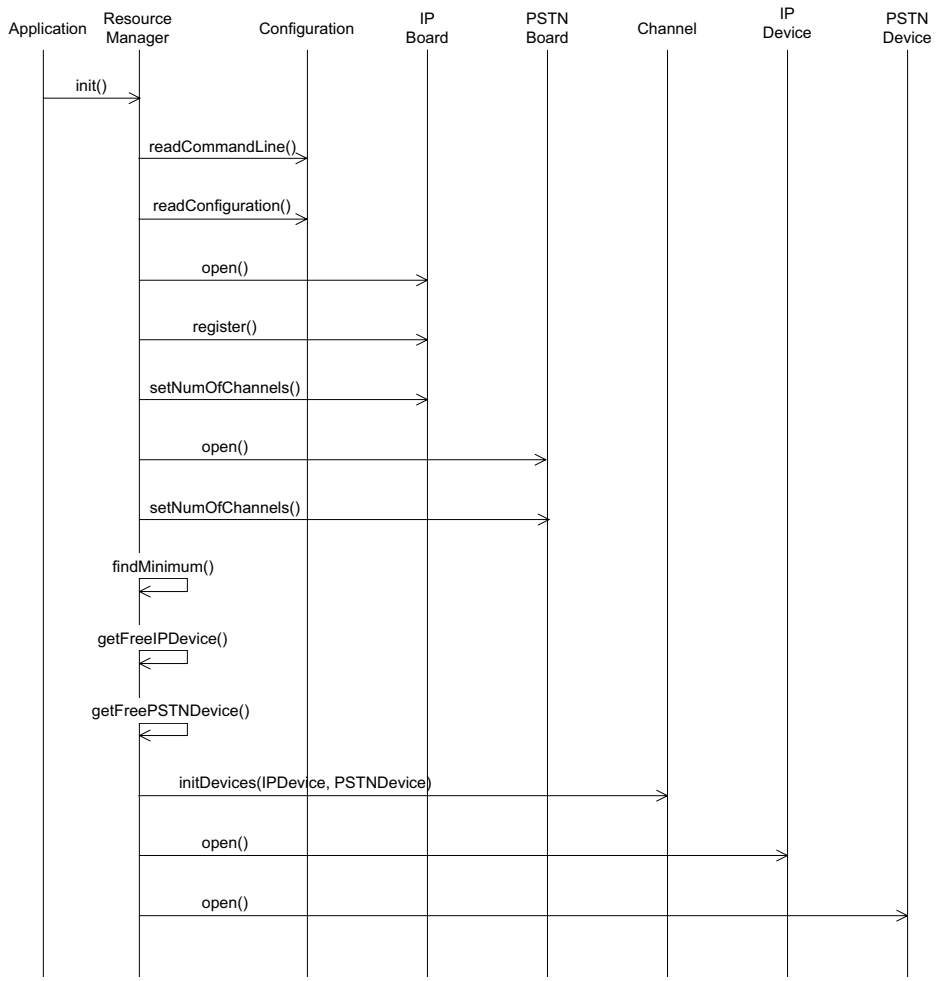
The threads are created as follows:

- The first (main) thread is created by the demo application to get the keyboard input.
- The second thread is an SRL thread, created as a result of the demo application calling **sr_enblhdlr( )** in Windows. In Linux, the thread must be explicitly created. All Global Call events are received through the SRL.

5.4 Initialization

This section describes the demo initialization as shown in Figure 4.

Figure 4. IP Gateway (Global Call) Object Oriented System Initialization



The application **main( )** function calls the **init( )** function, which does the following:

1. Calls **resourceManager.configure( )** to read the configuration file and command line options and prints the configuration
2. Calls **resourceManager.getChannelsNum( )** to get the number of channels defined by the user in the configuration file or command line -n switch.
3. Calls **gc_Start( )** to open all configured, call control libraries
4. Calls **printAllLibs( )** to print library status (open or failed).

5. Sets-up the callback handler, **PDLsr_enbhdlr( )**. The callback handler handles events that it receives from the SRL library. For more details see Section 5.5.3, "Handling SRL Events", on page 37.
6. Calls **resourceManager.init( )** to get the resources available in the system:
   a. Gets the number of IP channels in the system
   b. Gets the number of PSTN channels in the system
   c. Finds the minimum between the system channels and the user request
7. Looks for a free IP device and returns a pointer to it
8. Opens the IP device and if the open succeeds returns a pointer to it
9. Looks for a free PSTN device and returns a pointer to it
10. Opens the PSTN device and if the open success returns a pointer to it
11. Initializes the devices on the channel
12. The application **main( )** function calls **waitForKey( )**, to receive keyboard input.

5.5 Event Handling

This section describes how the IP Gateway (Global Call) Object Oriented demo handles events. It contains the following topics:

- Event Mechanism
- Handling Keyboard Input Events
- Handling SRL Events
- Handling Application Exit Events

5.5.1 Event Mechanism

The IP Gateway (Global Call) Object Oriented demo uses the SRL mechanism to retrieve events. When an event occurs, SRL calls event handlers automatically. All events are received by the SRL and then passed to the **callback_hdlr( )** function for handling.

In the initialization phase of the demo the **init( )** function sets up the call-back handler, by calling **PDLsr_enbhdlr( )**.

Refer to Chapter 6, "Demo State Machines" for more detailed event handling information.

5.5.2 Handling Keyboard Input Events

There is an endless loop **{while(1)}** in the **main( )** function in the *Main.cpp* file. In that loop, the application waits forever for a keyboard event by calling the **waitForKey( )** function. The event must be handled immediately and event-specific information should be retrieved before the next call to **waitForKey( )**.

When the next event occurs or when a time-out is reached, the **waitForKey( )** returns and the call-back handler function is called automatically.

5.5.3 Handling SRL Events

When the R4/Global Call event is received, the application performs the following:

1. Gets the event device handle, by calling **PDLsr_getevtdev( )**
2. Gets the channel number related to the event, from the global array (HandleToChannel[ ])
3. Updates the METAEVENT structure by calling **gc_GetMetaEvent( )**
4. Gets the event type, by calling **PDLsr_getevttype( )**

5.5.4 Handling Application Exit Events

Normal application exit events don't enter the SRL. The **main( )** function calls **PDLSetApplicationExitPath( )** before initialization. In Linux, this function sets the signals (SIGINT, SIGTERM, SIGABRT) for making the appropriate exit from the application. In Windows, this function enables the detection of CTRL_CLOSE_EVENT (closing the window).

Demo State Machines 6

This chapter discusses the IP Gateway (Global Call) Object Oriented state machines. It contains the following topics:

6.1 GWCall State Machine - Inbound Call from IP

This section describes the state machine for an inbound call from the IP. It contains the following topics:

- GWCall State Machine Description - Inbound from IP
- GWCall::callNull State
- GWCall::callGetIPInfo
- GWCall::callIPOffered
- GWCall::callIPAccepting
- GWCall::callPSTNConnected
- GWCall::callConnected
- GWCall::callDropping
- GWCall::callReleasing

6.1.1 GWCall State Machine Description - Inbound from IP

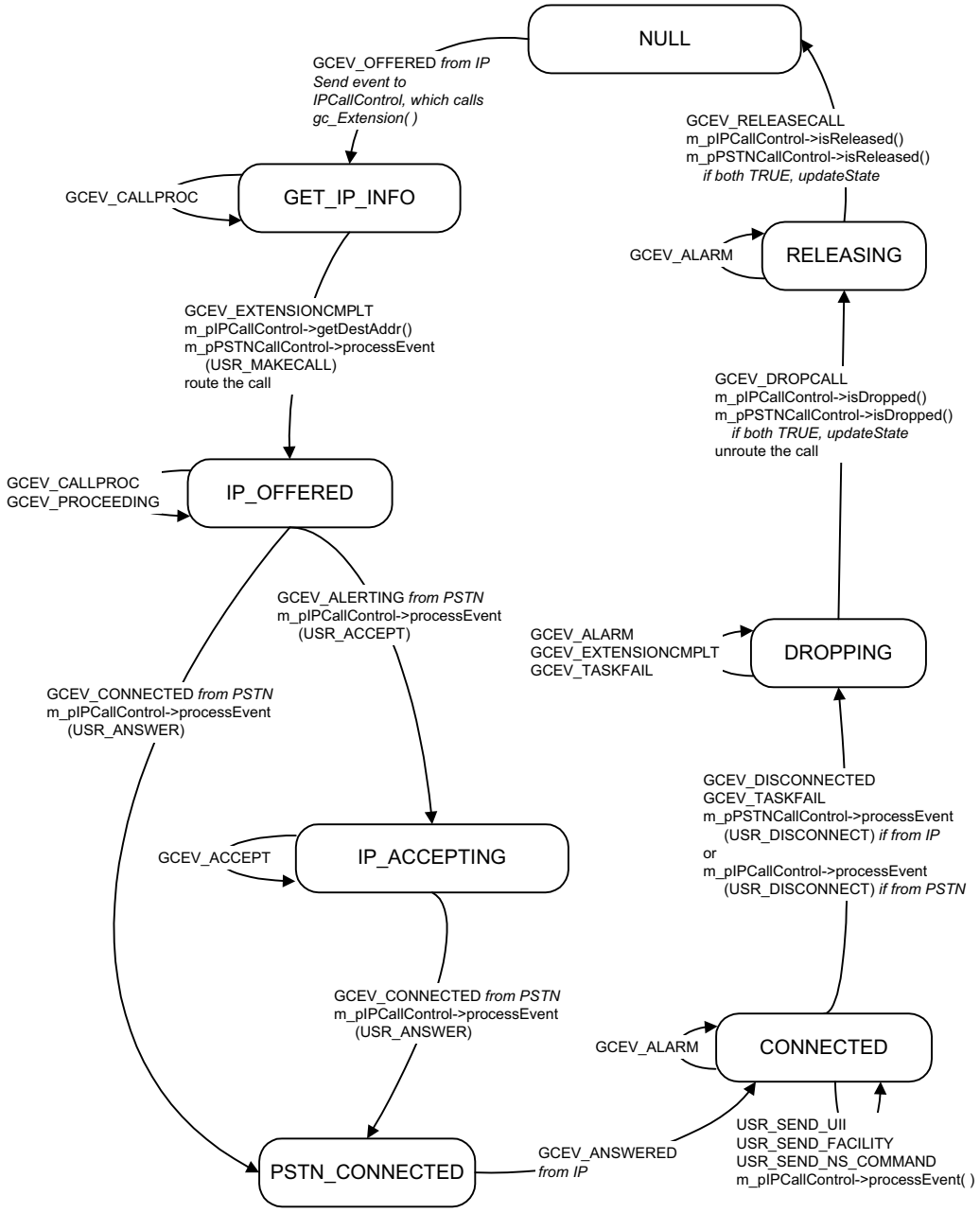
All channels are initialized to the NULL state upon application start.

As soon as an event is received, the event type, the channel number, and the reason for the event (if there is one), are analyzed and the appropriate state machine function is called.

After all the operations are performed within the channel's event state, the state machine function is updated.

The following state diagram describes the call states for the GWCall class for an inbound call from the IP.

Figure 5. GWCall State Machine - Inbound Call from IP



6.1.2 GWCall::callNull State

The application waits in the gateNull state for a GCEV_OFFERED event from the IP. Upon receipt of the event it transfers the event to the IPCallControl module which calls **getIncomingCallInfo( )**,

which in turn calls **gc_Extension( )**. See Section 6.4.2, "IPCallControl::Null State", on page 53 for additional information. The call state transitions to callGetIPInfo.

If the application receives a GCEV_DISCONNECTED event, it sets the drop reason by calling **getResultValue( )** from the GCCallControl module. The application then calls **processEvent(USR_DISCONNECT)** from the IPCallControl module if the PSTN was the event source, or from the PSTNCallControl module if the IP was the event source. The call state transitions to callDropping.

6.1.3 GWCall::callGetIPInfo

The application waits for a GCEV_EXTENSIONCMPLT event. Upon receipt of the event, it calls **getDestAddr( )** from the IPCallControl module and calls **processEvent(USR_MAKECALL)** from the PSTNCallControl module. See Section 6.3, "PSTNCallControl State Machine", on page 47 for a description of the PSTNCallControl state machine. The application then routes the call. The state transitions to callIPOffered.

If the application receives a GCEV_CALLPROC event, it ignores the event and remains in the callGetIPInfo state.

6.1.4 GWCall::callIPOffered

The application waits for either a GCEV_ALERTING or a GCEV_CONNECTED event from the PSTN.

In the case of GCEV_ALERTING, the application calls **processEvent(USR_ACCEPT)** from the IPCallControl module and the call state transitions to callIPAccepting.

In the case of GCEV_CONNECTED, the application calls **processEvent(USR_ANSWER)** from the IPCallControl module and the call state transitions to callPSTNConnected.

If the application receives a GCEV_CALLPROC or a GCEV_PROCEEDING event, it ignores the event and remains in the callGetIPOffered state.

If the application receives a GCEV_DISCONNECTED or a GCEV_TASKFAIL event, it sets the drop reason by calling **getResultValue( )** from the GCCallControl module. The application then calls **processEvent(USR_DISCONNECT)** from the IPCallControl module if the PSTN was the event source, or from the PSTNCallControl module if the IP was the event source. The call state transitions to callDropping.

6.1.5 GWCall::callIPAccepting

The application waits for a GCEV_CONNECTED event from the PSTN. Upon receipt of the event it calls **processEvent(USR_ANSWER)** from the IPCallControl module and the call state transitions to callPSTNConnected.

If the application receives a GCEV_ACCEPT event, it ignores the event and remains in the callIPAccepting state.

If the application receives a GCEV_DISCONNECTED or a GCEV_TASKFAIL event, it sets the drop reason by calling **getResultValue( )** from the GCCallControl module. The application then calls **processEvent(USR_DISCONNECT)** from the IPCallControl module if the PSTN was the event source, or from the PSTNCallControl module if the IP was the event source. The call state transitions to callDropping.

6.1.6 GWCall::callPSTNConnected

The application waits for a GCEV_ANSWERED event from the IP. Upon receipt of the event, the call state transitions to callConnected.

If the application receives a GCEV_DISCONNECTED or a GCEV_TASKFAIL event, it sets the drop reason by calling **getResultValue( )** from the GCCallControl module. The application then calls **processEvent(USR_DISCONNECT)** from the IPCallControl module if the PSTN was the event source, or from the PSTNCallControl module if the IP was the event source. The call state transitions to callDropping.

6.1.7 GWCall::callConnected

The application waits for a GCEV_DISCONNECTED or GCEV_TASKFAIL event. Upon receipt of either event, it sets the drop reason by calling **getResultValue( )** from the GCCallControl module. The application then calls **processEvent(USR_DISCONNECT)** from the IPCallControl module if the PSTN was the event source, or from the PSTNCallControl module if the IP was the event source. The call state transitions to callDropping.

The caller may also use the keyboard to send a User Input Indication, Facility Message, or a Non-standard Command. The application receives the appropriate event (USR_SEND_UII, USR_SEND_FACILITY, USR_SEND_NS_COMMAND) and calls **processEvent( )** from the IPCallControl module. The call state remains in the callConnected state.

6.1.8 GWCall::callDropping

The application waits for a GCEV_DROPCALL event. Upon receipt of the event, it calls **isDropped( )** from both the IPCallControl and PSTNCallControl modules. If both sides return TRUE, the call state transitions to callReleasing and the application unroutes the call.

If the application receives a GCEV_ALARM, GCEV_EXTENSIONCMPLT, or GCEV_TASKFAIL event, it ignores the event and remains in the callDropping state.

6.1.9 GWCall::callReleasing

The application waits for a GCEV_RELEASECALL event. Upon receipt of the event, it calls **isReleased( )** from both the IPCallControl and PSTNCallControl modules. If both sides return TRUE, the call state transitions to callNull.

If the application receives a GCEV_ALARM event, it ignores the event and remains in the callReleasing state.

6.2 GWCall State Machine - Inbound Call from PSTN

This section describes the state machine for an inbound call from the PSTN. It contains the following topics:

6.2.1 GWCall State Machine Description - Inbound from PSTN

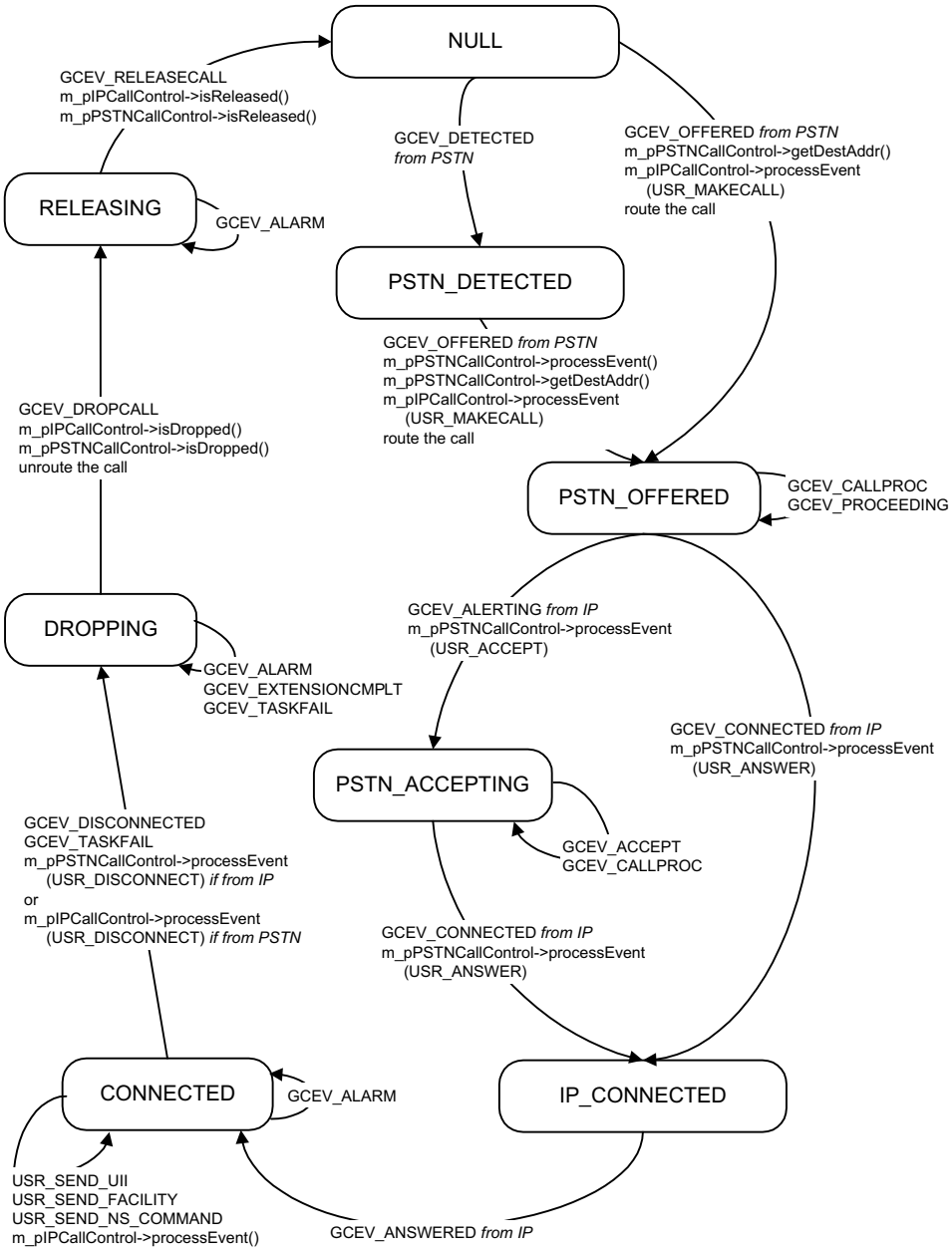
All channels are initialized to the NULL state upon application start.

As soon as an event is received, the event type, the channel number, and the reason for the event (if there is one), are analyzed and the appropriate state machine function is called.

After all the operations are performed within the channel's event state, the state machine function is updated.

The following state diagram describes the call states for the GWCall class for an inbound call from the PSTN.

Figure 6. GWCall State Machine - Inbound Call from PSTN



6.2.2 GWCall::callNull State

The application waits in the gateNull state for an GCEV_OFFERED event from the PSTN. Upon receipt of the event, the application sets the remote phone number for making the call by calling **getDestAddr( )** from the PSTNCallControl module. It then calls

processEvent(USR_MAKECALL) from the IPCallControl module. The application routes the call and the call state transitions to callPSTNOffered.

If the application receives a GCEV_DETECTED event, the call state transitions to callPSTNDetected.

If the application receives a GCEV_DISCONNECTED event, it sets the drop reason by calling **getResultValue( )** from the GCCallControl module. The application then calls **processEvent(USR_DISCONNECT)** from the IPCallControl module if the PSTN was the event source, or from the PSTNCallControl module if the IP was the event source. The call state transitions to callDropping.

6.2.3 GWCall::callPSTNDetected

The application waits for a GCEV_OFFERED event from the PSTN. Upon receipt of the event, the application calls **processEvent( )** from the PSTNCallControl module. The application then sets the remote phone number for making the call by calling **getDestAddr( )** from the PSTNCallControl module. It then calls **processEvent(USR_MAKECALL)** from the IPCallControl module. The application routes the call and the call state transitions to callPSTNOffered.

6.2.4 GWCall::callPSTNOffered

The application waits for either a GCEV_CONNECTED or a GCEV_ALERTING event from the IP.

In the case of a GCEV_CONNECTED event, the application calls **processEvent(USR_ANSWER)** from the PSTNCallControl module and the call state transitions to callIPConnected.

In the case of a GCEV_ALERTING event, the application calls **processEvent(USR_ACCEPT)** from the PSTNCallControl module and the call state transitions to callPSTNAccepting.

If the application receives a GCEV_CALLPROC or a GCEV_PROCEEDING event, it ignores the event and remains in the callPSTNOffered state.

If the application receives a GCEV_DISCONNECTED or a GCEV_TASKFAIL event, it sets the drop reason by calling **getResultValue( )** from the GCCallControl module. The application then calls **processEvent(USR_DISCONNECT)** from the IPCallControl module if the PSTN was the event source, or from the PSTNCallControl module if the IP was the event source. The call state transitions to callDropping.

6.2.5 GWCall::callPSTNAccepting

The application waits for a GCEV_CONNECTED event from the IP. Upon receiving the event it calls **processEvent(USR_ANSWER)** from the PSTNCallControl module and the call state transitions to callIPConnected.

If the application receives a GCEV_CALLPROC or a GCEV_ACCEPT event, it ignores the event and remains in the callPSTNAccepting state.

If the application receives a GCEV_DISCONNECTED or a GCEV_TASKFAIL event, it sets the drop reason by calling **getResultValue( )** from the GCCallControl module. The application then calls **processEvent(USR_DISCONNECT)** from the IPCallControl module if the PSTN was the event source, or from the PSTNCallControl module if the IP was the event source. The call state transitions to callDropping.

6.2.6 GWCall::callIPConnected

The application waits for a GCEV_ANSWERED from the PSTN. Upon receipt of the event the call state transitions to callConnected.

If the application receives a GCEV_EXTENSION event, it ignores the event and remains in the callIPConnected state.

The caller may also use the keyboard to send a User Input Indication, Facility Message, or a Non-standard Command. The application receives the appropriate event (USR_SEND_UII, USR_SEND_FACILITY, USR_SEND_NS_COMMAND) and calls **processEvent( )** from the IPCallControl module. The call state remains in the callIPConnected state.

If the application receives a GCEV_DISCONNECTED or a GCEV_TASKFAIL event, it sets the drop reason by calling **getResultValue( )** from the GCCallControl module. The application then calls **processEvent(USR_DISCONNECT)** from the IPCallControl module if the PSTN was the event source, or from the PSTNCallControl module if the IP was the event source. The call state transitions to callDropping.

6.2.7 GWCall::callConnected

The application waits for a GCEV_DISCONNECTED or GCEV_TASKFAIL event. Upon receipt of either event, it sets the drop reason by calling **getResultValue( )** from the GCCallControl module. The application then calls **processEvent(USR_DISCONNECT)** from the IPCallControl module if the PSTN was the event source, or from the PSTNCallControl module if the IP was the event source. The call state transitions to callDropping.

The caller may also use the keyboard to send a User Input Indication, Facility Message, or a Non-standard Command. The application receives the appropriate event (USR_SEND_UII, USR_SEND_FACILITY, USR_SEND_NS_COMMAND) and calls **processEvent( )** from the IPCallControl module. The call state remains in the callConnected state.

6.2.8 GWCall::callDropping

The application waits for a GCEV_DROPCALL event. Upon receipt of the event, it calls **isDropped( )** from both the IPCallControl and PSTNCallControl modules. If both sides return TRUE, the call state transitions to callReleasing and the application unroutes the call.

If the application receives a GCEV_ALARM, GCEV_EXTENSIONCMPLT, or GCEV_TASKFAIL event, it ignores the event and remains in the callDropping state.

6.2.9 GWCall::callReleasing

The application waits for a GCEV_RELEASECALL event. Upon receipt of the event, it calls **isReleased( )** from both the IPCallControl and PSTNCallControl modules. If both sides return TRUE, the call state transitions to callNull.

If the application receives a GCEV_ALARM event, it ignores the event and remains in the callReleasing state.

6.3 PSTNCallControl State Machine

This section describes the PSTNCallControl state machine. It contains the following topics:

6.3.1 PSTNCallControl State Machine Description

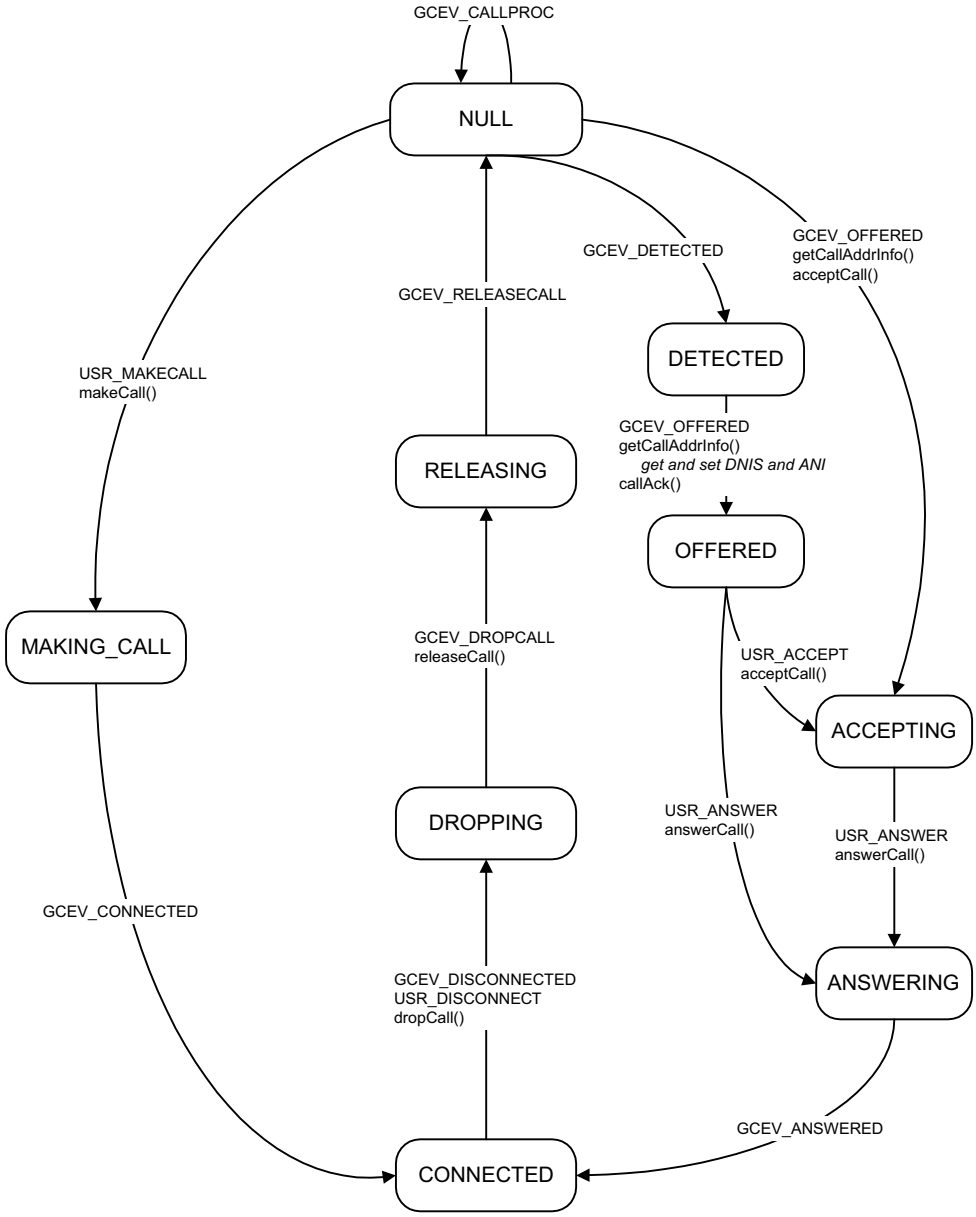
All channels are initialized to the NULL state upon application start.

As soon as an event is received, the event type, the channel number, and the reason for the event (if there is one), are analyzed and the appropriate state machine function is called.

After all the operations are performed within the channel's event state, the state machine function is updated.

The following state diagram describes the call states for the PSTNCallControl class.

Figure 7. PSTNCallControl State Machine



6.3.2 PSTNCallControl::Null State

The application waits for a GCEV_DETECTED, GCEV_OFFERED, or USR_MAKECALL event.

In the case of a GCEV_DETECTED event, the state transitions to the Detected state.

In the case of a GCEV_OFFERED event, the application calls **getCallAddrInfo( )** to get and set DNIS and ANI and then calls **acceptCall( )**. The state transitions to the Accepting state.

In the case of a USR_MAKECALL event, the application calls **makeCall( )** and the state transitions to makingCall.

If the application receives a GCEV_DISCONNECTED event, it calls **getResultValue( )** to get the disconnect reason and then calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

If the application receives a USR_DISCONNECT event it retrieves the DropReason from the EventData and calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

6.3.3 PSTNCallControl::Detected State

The application waits for a GCEV_OFFERED event. Upon receipt of the event, it calls **getCallAddrInfo( )** to get and set DNIS and ANI and then calls **callAck( )**. The state transitions to the Offered state.

If the application receives a GCEV_DISCONNECTED event, it calls **getResultValue( )** to get the disconnect reason and then calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the call state transitions to Init. Otherwise, the state transitions to Dropping.

If the application receives a USR_DISCONNECT event it retrieves the DropReason from the EventData and calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

6.3.4 PSTNCallControl::Offered State

The application waits for a USR_ACCEPT or USR_ANSWER event.

In the case of USR_ACCEPT, the application calls **acceptCall( )** and the state transitions to Accepting.

In the case of USR_ANSWER, the application calls **answerCall( )** and the state transitions to Answering.

If the application receives a GCEV_DISCONNECTED event, it calls **getResultValue( )** to get the disconnect reason and then calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

If the application receives a USR_DISCONNECT event it retrieves the DropReason from the EventData and calls **dropCall( )**. If the **dropCall( )** function fails, the application calls

resetLineDevice( ) to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

6.3.5 PSTNCallControl::Accepting State

The application waits for a USR_ANSWER event. Upon receipt of the event, it calls **answerCall( )** and the state transitions to Answering.

If the application receives a GCEV_ACCEPT event, it ignores the event and remains in the Accepting state.

If the application receives a GCEV_DISCONNECTED or GCEV_TASKFAIL event, it calls **getResultValue( )** to get the disconnect reason and then calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

If the application receives a USR_DISCONNECT event it retrieves the DropReason from the EventData and calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

6.3.6 PSTNCallControl::Answering State

The application waits for a GCEV_ANSWERED event. Upon receipt of the event, the state transitions to Connected.

If the application receives a GCEV_DISCONNECTED or GCEV_TASKFAIL event, it calls **getResultValue( )** to get the disconnect reason and then calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

If the application receives a USR_DISCONNECT event it retrieves the DropReason from the EventData and calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

6.3.7 PSTNCallControl::makingCall State

The application waits for a GCEV_CONNECTED event. Upon receipt of the event, the state transitions to Connected.

If the application receives a GCEV_ALERTING or GCEV_PROCEEDING event, it ignores the event and remains in the makingCall state.

If the application receives a GCEV_DISCONNECTED event, it calls **getResultValue( )** to get the disconnect reason and then calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

If the application receives a USR_DISCONNECT event it retrieves the DropReason from the EventData and calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

6.3.8 PSTNCallControl::Connected State

The application waits for a GCEV_DISCONNECTED or USR_DISCONNECT event. In the case of GCEV_DISCONNECTED, the application calls **getResultValue( )** to get the disconnect reason and then calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

In the case of USR_DISCONNECT, the application retrieves the DropReason from the EventData and calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

6.3.9 PSTNCallControl::Dropping State

The application waits for a GCEV_DROPCALL event. Upon receipt of the event, it calls **releaseCall( )** and the state transitions to Releasing.

6.3.10 PSTNCallControl::Releasing State

The application waits for a GCEV_RELEASECALL event. Upon receipt of the event, the state transitions to Null.

6.4 IPCallControl State Machine

This section describes the PSTNCallControl state machine. It contains the following topics:

6.4.1 IPCallControl State Machine Description

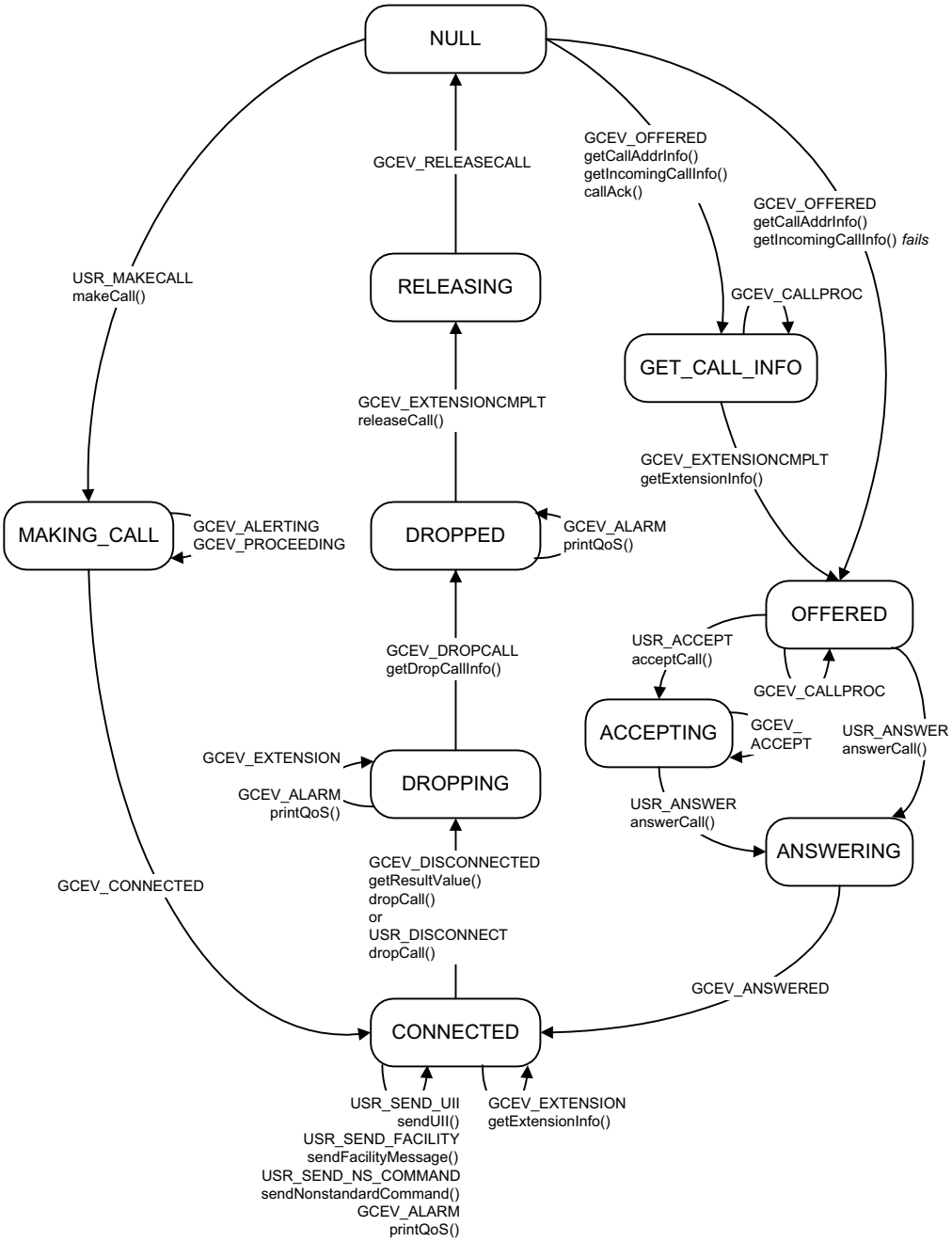
All channels are initialized to the NULL state upon application start.

As soon as an event is received, the event type, the channel number, and the reason for the event (if there is one), are analyzed and the appropriate state machine function is called.

After all the operations are performed within the channel's event state, the state machine function is updated.

The following state diagram describes the call states for the IPCallControl class.

Figure 8. IPCallControl State Machine



6.4.2 IPCallControl::Null State

The application waits for a GCEV_OFFERED or USR_MAKECALL event. In the case of GCEV_OFFERED, the application calls **getCallAddrInfo( )** to get the destination and originator

from the IP message and set the member variables. The application then calls **getIncomingCallInfo( )** which calls **gc_Extension( )** to request extension information from the IP. If the function fails, the application does not wait for the complete message and the state transitions to Connected. Otherwise, the application calls **callAck( )** and the state transitions to getCallInfo.

In the case of USR_MAKECALL, the application calls **makeCall( )** and the state transitions to makingCall.

If the application receives a GCEV_DISCONNECTED event, it calls **getResultValue( )** to get the disconnect reason and then calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

If the application receives a USR_DISCONNECT event it retrieves the DropReason from the EventData and calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

6.4.3 IPCallControl::getCallInfo State

The application waits for a GCEV_EXTENSIONCMPLT event. The application calls **getExtensionInfo( )** and the state transitions to Offered.

If the application receives a GCEV_CALLPROC event, acknowledging the **callAck( )** function, it ignores the event and remains in the getCallInfo state.

If the application receives a GCEV_DISCONNECTED or GCEV_TASKFAIL event, it calls **getResultValue( )** to get the disconnect reason and then calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

If the application receives a USR_DISCONNECT event it retrieves the DropReason from the EventData and calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

6.4.4 IPCallControl::offered State

The application waits for a USR_ACCEPT or USR_ANSWER event. In the case of USR_ACCEPT, the application calls **acceptCall( )** and the state transitions to Accepting.

In the case of USR_ANSWER, the application calls **answerCall( )** and the state transitions to Answering.

If the application receives a GCEV_CALLPROC event, acknowledging the **callAck( )** function, it ignores the event and remains in the Offered state.

If the application receives a GCEV_DISCONNECTED or GCEV_TASKFAIL event, it calls **getResultValue( )** to get the disconnect reason and then calls **dropCall( )**. If the **dropCall( )**

function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

If the application receives a USR_DISCONNECT event it retrieves the DropReason from the EventData and calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

6.4.5 IPCallControl::accepting State

The application waits for a USR_ANSWER event. Upon receipt of the event, it calls **answerCall( )** and the state transitions to Answering.

If the application receives a GCEV_ACCEPT event, it ignores the event and remains in the Accepting state.

If the application receives a GCEV_DISCONNECTED or GCEV_TASKFAIL event, it calls **getResultValue( )** to get the disconnect reason and then calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

If the application receives a USR_DISCONNECT event it retrieves the DropReason from the EventData and calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

6.4.6 IPCallControl::answering State

The application waits for a GCEV_ANSWERED event. Upon receipt of the event the state transitions to Connected.

If the application receives a GCEV_DISCONNECTED or GCEV_TASKFAIL event, it calls **getResultValue( )** to get the disconnect reason and then calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

If the application receives a USR_DISCONNECT event it retrieves the DropReason from the EventData and calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

6.4.7 IPCallControl::makingCall State

The application waits for a GCEV_CONNECTED event. Upon receipt of the event, the state transitions to Connected.

If the application receives a GCEV_ALERTING or GCEV_PROCEEDING event, it ignores the event and remains in the makingCall state.

If the application receives a GCEV_DISCONNECTED or GCEV_TASKFAIL event, it calls **getResultValue( )** to get the disconnect reason and then calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

If the application receives a USR_DISCONNECT event it retrieves the DropReason from the EventData and calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

6.4.8 IPCallControl::connected State

The application waits for a GCEV_DISCONNECTED or USR_DISCONNECT event. In the case of GCEV_DISCONNECTED, the application calls **getResultValue( )** to get the disconnect reason and then calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

In the case of USR_DISCONNECT, the application retrieves the DropReason from the EventData and calls **dropCall( )**. If the **dropCall( )** function fails, the application calls **resetLineDevice( )** to reset the line device and the state transitions to Init. Otherwise, the state transitions to Dropping.

If the application receives a GCEV_EXTENSION event, it calls **getExtensionInfo( )** and remains in the Connected state.

If the application receives a USR_SEND_UII event, it calls **sendUII( )** and remains in the Connected state.

If the application receives a USR_SEND_FACILITY event, it calls **sendFacilityMessage( )** and remains in the Connected state.

If the application receives a USR_SEND_NS_COMMAND event, it calls **sendNonstandardCommand( )** and remains in the Connected state.

If the application receives a GCEV_ALARM event, it calls **printQoS( )** and remains in the Connected state.

6.4.9 IPCallControl::dropping State

The application waits for a GCEV_DROPCALL event. Upon receipt of the event, it calls **getDropCallInfo( )**, which calls **gc_Extension( )** and the state transitions to Dropped.

If the application receives a GCEV_EXTENSION event, it ignores the event and remains in the Dropping state.

If the application receives a GCEV_ALARM event, it calls **printQoS( )** and remains in the Dropping state.

6.4.10 IPCallControl::dropped State

The application waits for a GCEV_EXTENSIONCMPLT event. Upon receipt of the event it calls **releaseCall( )** and the state transitions to Releasing.

If the application receives a GCEV_ALARM event, it calls **printQoS( )** and remains in the Dropped state.

6.4.11 IPCallControl::releasing State

The application waits for a GCEV_RELEASECALL event. Upon receipt of the event the state transitions to Null.

Glossary

Asynchronous mode: An operating mode of certain DM3 kernel or host library function calls. Asynchronous mode is a non-blocking mode that is typically used when a function involves operation on a remote processor (e.g., a host library function that initiates a kernel operation on a DM3 platform's Control Processor) or when data movement via the MMA and global memory is required. In asynchronous mode a value signifying "pending" status is immediately returned to the calling function, and the actual completion or failure of the operation is reported to the caller via a DM3 result message.

Codec: see COder/DECoder

COder/DECoder: A circuit used on Dialogic boards to convert analog voice data to digital and digital voice data to analog audio.

Computer Telephony (CT): Adding computer intelligence to the making, receiving, and managing of telephone calls.

Control Processor (CP): A processor that is present on the motherboard of every DM3 platform for the management of the SCbus/CT bus, for host communication configuration, and to implement some of the features of DM3 Resources. All current DM3 platforms use an Intel i960 as the Control Processor.

DM3 kernel: Firmware that is present on each processor on a DM3 platform to support configuration management, host communication, inter processor communication and control, and SCSA firmware services among others. There may be more than one version of the DM3 kernel firmware depending on the type of processor (e.g. CP vs. DSP or RISC signal processor) and the intended application of each processor.

DM3 load module: A loadable block of executable firmware for a particular processor on a DM3 platform. A DM3 Resource may be packaged as one or more DM3 load modules.

DTMF: See Dual-Tone Multi-Frequency

Dual-Tone Multi-Frequency: A way of signaling consisting of a push-button or touch-tone dial that sends out a sound consisting of two discrete tones that are picked up and interpreted by telephone switches (either PBXs or central offices).

Emitting Gateway: called by a G3FE. It initiates IFT service for the calling G3FE and connects to a Receiving Gateway.

E1: The 2.048 Mbps digital carrier system common in Europe.

FCD file: An ASCII file that lists any non-default parameter settings that are necessary to configure a DM3 hardware/firmware product for a particular feature set. The downloader utility reads this file, and for each parameter listed generates and sends the DM3 message necessary to set that parameter value.

Frame: A set of SCbus/CT bus timeslots which are grouped together for synchronization purposes. The period of a frame is fixed (at 125 µsec) so that the number of time slots per frame depends on the SCbus/CT bus data rate. In

the context of DSP programming (e.g. DM3 component development), the period defined by the sample rate of the signal data.

G3FE: Group 3 Fax Equipment. A traditional fax machine with analog PSTN interface.

Gatekeeper: An H.323 entity on the Internet that provides address translation and control access to the network for H.323 Terminals and Gateways. The Gatekeeper may also provide other services to the H.323 terminals and Gateways, such as bandwidth management and locating Gateways.

Gateway: An H.323 Gateway (GW) is an endpoint on the Internet which provides for real-time, two-way communications between H.323 Terminals on the Network and other ITU Terminals on a wide area network, or to IP Telephony clients.

H.323: A set of International Telecommunication Union (ITU) standards that define a framework for the transmission of real-time voice communications through Internet protocol (IP)-based packet-switched networks. The H.323 standards define a gateway and a gatekeeper for customers who need their existing IP networks to support voice communications.

IAF: Internet Aware Fax. The combination of a G3FE and a T.38 gateway.

IFP: Internet Facsimile Protocol

IFT: Internet Facsimile Transfer

International Telecommunications Union (ITU): An organization established by the United Nations to set telecommunications standards, allocate frequencies to various uses, and hold trade shows every four years.

Internet: An inter-network of networks interconnected by bridges or routers. LANs described in H.323 may be considered part of such inter-networks.

Internet Protocol (IP): The network layer protocol of the transmission control protocol/Internet protocol (TCP/IP) suite. Defined in STD 5, Request for Comments (RFC) 791. It is a connectionless, best-effort packet switching protocol.

Internet Service Provider (ISP): A vendor who provides direct access to the Internet.

Internet Telephony: The transmission of voice over an Internet Protocol (IP) network. Also called Voice over IP (VoIP), IP telephony enables users to make telephone calls over the Internet, intranets, or private Local Area Networks (LANs) and Wide Area Networks (WANs) that use the Transmission Control Protocol/Internet Protocol (TCP/IP).

ITU: See International Telecommunications Union.

Jitter: The deviation of a transmission signal in time or phase. It can introduce errors and loss of synchronization in high-speed synchronous communications.

NIC (Network Interface Card): Adapter card inserted into computer that contains necessary software and electronics to enable station to communicate over network.

Parameter: A type of datum that is passed to or from a component via a DM3 message. Parameters generally have two elements, a type and a specific value. Parameters passed to a component affects the operational characteristics of the component, and parameters passed from a component can be used to report the operating state of the component. As an example, the standard Player component accepts a parameter called ParmDuration, which specifies the length of the file that is being played back, and can report its current state (e.g. playing, paused, etc.) via a parameter called ParmState.

PCD file: An ASCII text file that contains product or platform configuration description information that is used by the DM3 downloader utility program. Each of these files identifies the hardware configuration and firmware modules that make up a specific hardware/firmware product. Each type of DM3-based product used in a system requires a product-specific PCD file.

PSTN: see Public Switched Telephone Network

Public Switched Telephone Network: The telecommunications network commonly accessed by standard telephones, key systems, Private Branch Exchange (PBX) trunks and data equipment.

Receiving Gateway: accepts a connection from an Emitting Gateway. It calls a G3FE and provides IFT service to the called G3FE.

Reliable Channel: A transport connection used for reliable transmission of an information stream from its source to one or more destinations.

Reliable Transmission: Transmission of messages from a sender to a receiver using connection-mode data transmission. The transmission service guarantees sequenced, error-free, flow-controlled transmission of messages to the receiver for the duration of the transport connection.

RTCP: Real Time Control Protocol

RTP: Real Time Protocol

SCbus: The standard bus for communication within a SCSA node. The architecture of SCbus includes a 16-wire TDM data bus that operates at 2, 4 or 8 Mbps and a serial message bus for control and signaling. DM3 platforms provide an SCbus interface for interconnection of multiple DM3 platforms, or connection to other SCSA-compatible hardware. The DM3 platform supports timeslot bundling for high bandwidth, and can access up to 256 of the 2048 SCbus timeslots via two SC4000 ASICs.

Signal Processor (SP): An embedded processor on a DM3 platform that is used to execute signal processing algorithms. The DM3 architecture supports multiple types of SPs, including RISC processors as well as general purpose DSPs, and platforms may be configured with a mixed complement of SP types.

SIP: Session Initiation Protocol: an Internet standard specified by the Internet Engineering Task Force (IETF) in RFC 2543. SIP is used to initiate, manage, and terminate interactive sessions between one or more users on the Internet.

Synchronous mode: An operating mode of certain DM3 kernel or host library function calls. Synchronous mode is a blocking mode that is typically used only when a function executes on the local processor using data that is also local to the processor (i.e. that does not require any data movement via the MMA).

T1: A digital transmission link with a capacity of 1.544 Mbps used in North America. Typically channeled into 24 digital subscriber level zeros (DS0s), each capable of carrying a single voice conversation or data stream. T1 uses two pairs of twisted pair wires.

TCP: see Transmission Control Protocol

Terminal: An H.323 Terminal is an endpoint on the local area network which provides for real-time, two-way communications with another H.323 terminal, Gateway, or Multipoint Control Unit. This communication consists of control, indications, audio, moving color video pictures, and/or data between the two terminals. A terminal may provide speech only, speech and data, speech and video, or speech, data, and video.

Transmission Control Protocol: The TCP/IP standard transport level protocol that provides the reliable, full duplex, stream service on which many application protocols depend. TCP allows a process on one machine to send a stream of data to a process on another. It is connection-oriented in the sense that before transmitting data, participants must establish a connection.

UDP: see User Datagram Protocol

UDPTL: Facsimile UDP Transport Layer protocol

User Datagram Protocol: The TCP/IP standard protocol that allows an application program on one machine to send a datagram to an application program on another machine. Conceptually, the important difference between UDP datagrams and IP datagrams is that UDP includes a protocol port number, allowing the sender to distinguish among multiple destinations on the remote machine.

VAD: Voice Activity Detection

Index