IVR SDK 8.1 C

Developer's Guide

About Genesys

Genesys is the world's leading provider of customer service and contact center software - with more than 4,000 customers in 80 countries. Drawing on its more than 20 years of customer service innovation and experience, Genesys is uniquely positioned to help companies bring their people, insights and customer channels together to effectively drive today's customer conversation. Genesys software directs more than 100 million interactions every day, maximizing the value of customer engagement and differentiating the experience by driving personalization and multi-channel customer service - and extending customer service across the enterprise to optimize processes and the performance of customer-facing employees. Go to www.genesyslab.com for more information.

Each product has its own documentation for online viewing at the Genesys Technical Support website or on the Documentation Library DVD, which is available from Genesys upon request. For more information, contact your sales representative.

Notice

Although reasonable effort is made to ensure that the information in this document is complete and accurate at the time of release, Genesys Telecommunications Laboratories, Inc., cannot assume responsibility for any existing errors. Changes and/or corrections to the information contained in this document may be incorporated in future versions.

Your Responsibility for Your System's Security

You are responsible for the security of your system. Product administration to prevent unauthorized use is your responsibility. Your system administrator should read all documents provided with this product to fully understand the features available that reduce your risk of incurring charges for unlicensed use of Genesys products.

Trademarks

Genesys, the Genesys logo, and T-Server are registered trademarks of Genesys Telecommunications Laboratories, Inc. All other company names and logos may be trademarks or registered trademarks of their respective holders. © 2012 Genesys Telecommunications Laboratories, Inc. All rights reserved.

The Crystal monospace font is used by permission of Software Renovation Corporation, www.SoftwareRenovation.com.

Technical Support from VARs

If you have purchased support from a value-added reseller (VAR), please contact the VAR for technical support.

Technical Support from Genesys

If you have purchased support directly from Genesys, please contact Genesys Technical Support at the regional numbers provided on page 11. For complete contact information and procedures, refer to the ***Genesys Technical Support Guide***.

Ordering and Licensing Information

Complete information on ordering and licensing Genesys products can be found in the ***Genesys Licensing Guide.***

Released by

Genesys Telecommunications Laboratories, Inc. www.genesyslab.com

Document Version: 81sdk_dev_ivr-c_08-2012_v8.1.001.00

Table of Contents

Preface

Welcome to the *IVR SDK 8.1 C Developer's Guide.* This guide introduces you to the concepts, terminology, and procedures relevant to the Genesys IVR SDK C, the tool for building drivers that allow your IVR (Interactive Voice Response Unit) to communicate with the Genesys IVR Server.

This document is valid only for the 8.1 release of this product.

Note: For versions of this document created for other releases of this product, visit the Genesys Technical Support website, or request the Documentation Library DVD, which you can order by e-mail from Genesys Order Management at orderman@genesyslab.com.

This preface contains the following sections:

For information about related resources and about the conventions that are used in this document, see the supplementary material starting on page 91.

About IVR SDK C

In brief, this guide includes the following information:

- Definitions of an IVR driver, IVR Server, and IVR Library (I-Library)
- An explanation of the request and reply convention, with examples
- Library initialization and IVR Server connection
- How to work with version, telephony, and user data functions
- A digest of the IVR API
- How configuration data is used to direct processing

- Log files and setting log levels
- The DTD level and new features
- IVR Servers and load sharing
- The call state model
- XML escape characters
- The I-Library KeepAlive processing
- How response messages are processed
- The I-Library error codes for API request
- XML routing examples
- The 7.0 operation mode
- How I-Library handles outbound calls

New In Release 8.1

The following changes have been implemented in release 8.1:

- The ability to retrieve call information of type `UUID`.
- The ability to retrieve call information of type `ThirdPartyDN`.
- Optionally, call information of type `All` may now be configured on IVR Server to return either `UUID` or `ThirdPartyDN` (or both). By default, neither will be returned for an `All` request.
- Support for IPv6. For more information, refer to the *Framework 8.1 Deployment Guide.*

Intended Audience

This document is primarily intended for C programmers who must create an IVR driver to be integrated with a specific IVR system. It has been written with the assumption that you have a basic understanding of:

- Computer-telephony integration (CTI) concepts, processes, terminology, and applications
- Network design and operation
- Your own network configurations

You should also be familiar with:

- Genesys Framework architecture and functions
- Genesys IVR Server.
- Your IVR system.
- C programming language and the use of third-party libraries.

Usage Guidelines

The Genesys developer materials outlined in this document are intended to be used for the following purposes:

- Creation of contact center agent desktop applications associated with Genesys software implementations.
- Server-side integration between Genesys software and third-party software.
- Creation of a specialized client application specific to customer needs.

The Genesys software functions available for development are clearly documented. No undocumented functionality is to be utilized without the express written consent of Genesys.

The following Use Conditions apply in all cases for developers employing the Genesys developer materials outlined in this document:

1. Possession of interface documentation does not imply a right to use by a third party. Genesys conditions for use, as outlined below or in the *Genesys Developer Program Guide,* must be met.
2. This interface shall not be used unless the developer is a member in good standing of the Genesys Interacts program or has a valid Master Software License and Services Agreement with Genesys.
3. A developer shall not be entitled to use any licenses granted hereunder unless the developer's organization has met or obtained all prerequisite licensing and software as set out by Genesys.
4. A developer shall not be entitled to use any licenses granted hereunder if the developer's organization is delinquent in any payments or amounts owed to Genesys.
5. A developer shall not use the Genesys developer materials outlined in this document for any general application development purposes that are not associated with the above-mentioned intended purposes for the use of the Genesys developer materials outlined in this document.
6. A developer shall disclose the developer materials outlined in this document only to those employees who have a direct need to create, debug, and/or test one or more participant-specific objects and/or software files that access, communicate, or interoperate with the Genesys API.
7. The developed works and Genesys software running in conjunction with one another (hereinafter referred to together as the "integrated solutions") should not compromise data integrity. For example, if both the Genesys software and the integrated solutions can modify the same data, then modifications by either product must not circumvent the other product's data integrity rules. In addition, the integration should not cause duplicate

copies of data to exist in both participant and Genesys databases, unless it can be assured that data modifications propagate all copies within the time required by typical users.

8. The integrated solutions shall not compromise data or application security, access, or visibility restrictions that are enforced by either the Genesys software or the developed works.
9. The integrated solutions shall conform to design and implementation guidelines and restrictions described in the *Genesys Developer Program Guide* and Genesys software documentation. For example:
   a. The integration must use only published interfaces to access Genesys data.
   b. The integration shall not modify data in Genesys database tables directly using SQL.
   c. The integration shall not introduce database triggers or stored procedures that operate on Genesys database tables.

Any schema extension to Genesys database tables must be carried out using Genesys Developer software through documented methods and features.

The Genesys developer materials outlined in this document are not intended to be used for the creation of any product with functionality comparable to any Genesys products, including products similar or substantially similar to current Genesys general-availability, beta, and announced products.

Any attempt to use the Genesys developer materials outlined in this document or any Genesys Developer software contrary to this clause shall be deemed a material breach with immediate termination of this addendum, and Genesys shall be entitled to seek to protect its interests, including but not limited to, preliminary and permanent injunctive relief, as well as money damages.

Making Comments on This Document

If you especially like or dislike anything about this document, feel free to e-mail your comments to Techpubs.webadmin@genesyslab.com.

You can comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this document. Please limit your comments to the scope of this document only and to the way in which the information is presented. Contact your Genesys Account Representative or Genesys Technical Support if you have suggestions about the product itself.

When you send us comments, you grant Genesys a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

Contacting Genesys Technical Support

If you have purchased support directly from Genesys, contact Genesys Technical Support at the regional numbers below.

Note: The following contact information was correct at time of publication. For the most up-to-date contact information, see the Contact Information on the Tech Support website. Before contacting technical support, refer to the *Genesys Technical Support Guide* for complete contact information and procedures.

Genesys Technical Support Contact Information

Region	Telephone	E-Mail
North America	+888-369-5555 (toll-free) +506-674-6767	support@genesyslab.com
Latin America	+55-11-2948-5776	la-support@genesyslab.com
Europe, Middle East, and Africa	+44-(0)-127-645-7002	support@genesyslab.co.uk
Asia Pacific	+61-7-3368-6868	support@genesyslab.com.au
Japan	+81-3-6361-8950	support@genesyslab.co.jp
India	000-800-100-7136 (toll-free) +61-7-3368-6868	support@genesyslab.com.au

Document Change History

This is the first release of the *IVR SDK 8.1 C Developer's Guide.* In the future, this section will list topics that are new or that have changed significantly since the first release of this document.

Genesys

Chapter

1 How it Works

This chapter introduces essential concepts for developing an IVR (Interactive Voice Response) driver that uses the Genesys IVR Library (I-Library). This chapter has these sections:

Overview

An IVR driver provides an interface between an IVR system and the Genesys I-Library, which communicates with Genesys IVR Server.

Use standard C programming development tools to create an IVR driver that uses the API of the Genesys I-Library.

The names of configuration objects must match those stored in the Genesys Configuration Layer.

Architecture

An IVR system is a combination of software and hardware that provides interactive voice response functionality.

You can purchase an IVR driver software program from Genesys or develop it yourself, using the Genesys IVR SDK. Once in place, the IVR driver provides a brokering function between the IVR system and the Genesys I-Library.

The IVR Library (I-Library) software, in turn, provides the functionality that an IVR driver uses to communicate with a Genesys IVR Server on a given Local Area Network (LAN).

Communication between your driver and your IVR system is defined by your IVR vendor. Communication between your driver and the I-Library is within the same process space. Communication between the I-Library and the IVR Server is through TCP/IP sockets.

IVR Driver

You can design an IVR driver as three layers, one that interfaces with an IVR system, one that interfaces with the I-Library, and a middle layer (pad layer) that implements logic glue and any other functionality unrelated to the two interfaces.

The IVR system, typically through means of a script, communicates with the IVR driver, passing in requests for behavior and receiving data. This document does not discuss that interface. For that information, consult the documentation for your particular IVR system.

The IVR driver uses the I-Library API to pass IVR system requests to the IVR Server and a T-Server. This interface is the focus of this book.

IVR Library (I-Library)

The I-Library API interface comprises about 40 functions that can be grouped into the following categories:

- Initialization and deinitialization functions
- Connection and disconnection functions
- Basic utility functions
- Telephony functions
- Telephony
- Udata (User Data)
- Routing
- Statistics

Most functionality is implemented as a set of request functions that provide data through the use of a reply function. See Chapter 2, “Code Example One: Hello IVR World,” page 21, for a complete explanation of making requests and getting replies. See Chapter 5, “IVR API at a Glance,” page 59, for a digest of the API functions. See the online *HTML API Reference* in the documentation directory on the product CD for the most complete descriptions of the API. See the `interface.h` header file for types, enums, and defined constants.

IVR Server

The IVR SDK offers two interfaces for working with IVR Server, one for incoming C-based I-Library requests and the other for incoming XML-based requests.

Note: Even with C-based requests, the I-Library uses an XML interface that presents XML to the IVR Server. See Figure 1.

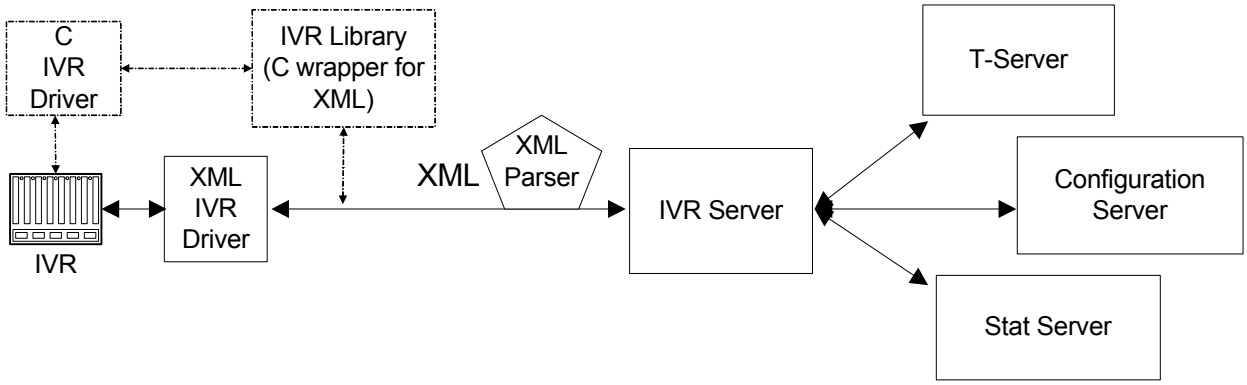

Figure 1: Genesys IVR Architecture

The I-Library handles requests for activity in an order-processing layer. Each request is stored as an order, with a unique order ID.

The driver uses request functions to pass in requests for data or some activity. The I-Library passes back the order ID for the request. The driver must use a reply function that specifies a particular order ID to get the results of a previous request. An order is valid for the duration of the call.

An IVR Server may service multiple clients, maintaining session information for each with respect to a session name passed in by the client. Each IVR system has to be configured separately with its own set of configuration parameters. The IVR driver must use this exact session name to control the particular IVR system.

There are multiple modes of operation for IVRs, each having a distinct impact on how you develop your driver. See the *IVR Interface Option 8.1 IVR Server System Administrator's Guide* for details about in-front, behind, and network modes of operation.

Development Requirements

To develop your own IVR driver, you must have a C compiler and an IVR with its development environment.

The Genesys IVR SDK provides you with the I-Library and its API, as represented in the `interface.h` header file, along with this *Developer's Guide* and the HTML API reference.

The I-Library files are as follows:

- On Windows: `ilib_SDK_MD.dll and ilib_SDK_MD.lib`
- On Solaris, Linux, and AIX: `libilib_SDK_32.so`

Deployment and Configuration

Your driver and the I-Library will be linked together in 6.5 operating mode and so will be on the same host.

Typically the IVR system software components are also loaded on the same host.

The IVR Servers and Genesys T-Servers are typically loaded on some other hosts on the same LAN.

The Genesys Configuration Layer stores common names and data for your driver as well as the IVR Server and T-Server with which your driver works. Be sure that the names your driver uses exactly match the names stored by the Configuration Layer, which are case sensitive.

The IVR driver must gather and pass in:

- The name of the `IVR` object in the Configuration Layer that contains the configuration data IVR Server needs to use for this driver. This `IVR` object name is also referred to as a *driver name*.
- The host name of the IVR Server.
- The TCP/IP port number that the IVR Server is using, the value for the `gli-server-address` option.

The IVR Server gathers the following information from the Configuration Layer:

- Resources assigned to each IVR system (port numbers and their associations with DNs).
- Log file parameters (the name of the file, its maximum length, number of backup copies, type of information, and so on).

Sockets, Ports, Channels, and DNs

The IVR Server is configured with a list of DNs and channels that correspond to a particular IVR. (See the *IVR Interface Option 8.1 IVR Server System Administrator's Guide* for more information.) The IVR Server associates the IVR channel numbers to the actual DNs. The association between channels and DNs is defined in the configuration source. Channels are treated as ports, specified as `ilPort` values.

Regardless of the number of ports in use, for a given IVR driver and IVR Server pair, there is only one socket connection between them. This socket is used to handle all the requests and events related to the activity on the IVR which flow through this IVR driver and I-Library.

Note: There is no need to open and close this socket with each request. In fact, the I-Library will prevent the driver from doing so. Once the driver starts, there is no way for it to ask the I-Library to disconnect from all IVR Servers with which it is communicating, and then restart communications with them.

T-Server Information

The IVR Server gets information from the T-Server interface in the form of events. A given incoming event replaces the previous event, and the IVR Server returns the latest event it has received to the IVR driver.

Miscellaneous Issues

For those IVRs that are designed to directly communicate with IVR Server using XML, there are certain functions and procedures that are included in the C SDK and may need to be duplicated.

Response Processing

The C SDK function interprets all messages that are received from IVR Server, and updates the request ID on a port-by-port basis. In order to achieve this, the following functionality is available:

- Interpreting all messages from IVR Server including format error checking.
- Correlating each message to the request that caused it.
- Formulating the appropriate response data for each message.
- Monitoring the TCP/IP socket to IVR Server for messages received.
- Handling error conditions from IVR Server.

- Handling post call requests such as the `GetCallInfo` last event.
- Maintaining the data until it is requested.
- Recognizing the duration of the call with which the data is associated, and expiring the data when appropriate.
- Tracking the state of the call to minimize traffic on the socket to IVR Server including routing sequences.
- Maintaining the last error that occurred on each port.

Load Sharing

Multiple IVR Servers can be used to handle call volume or for backup purposes. In order to achieve this, the following functionality is available:

- Associating all requests for a particular call with the IVR Server to which the `NewCall` was sent.
- Monitoring the TCP/IP socket to each IVR Server for connectivity.
- Balancing the call load between the load sharing IVR Servers using a modulo algorithm.

For detailed information about load sharing see Chapter 4, “Extended Functionality,” on page 39.

Connectivity to IVR Server

A TCP/IP socket communicates with each IVR Server that is used for call processing. In order to maintain this socket, the following functionality is available:

- Tracking and processing every successful login response to each login request.
- Handling any error conditions that occur for each login response.
- Monitoring for delays due to network inconsistencies or Wide Area Network (WAN) architecture.
- Tracking and processing messages split over TCP/IP packets.
- Tracking each socket, and reconnecting it if a disconnect occurs.
- Communicating with each IVR Server based on its Document Type Definition (DTD) level.
- KeepAlive integrity must be maintained. See “KeepAlive Processing” on page 45.

For details information about connectivity see Chapter 4, “Extended Functionality,” on page 39.

Configuration

A level of variability is required in order to adjust processing so that it conforms to the network environment and user preferences—including the ability to dynamically change these preferences. In order to provide the necessary level of variability, several configuration parameters need to be duplicated; and this duplicate information must be maintained for the duration of the process. All configuration data in Configuration Layer is available for this purpose. See “Configuration Data” on page 39 for more details.

API Processing

Before you send a request to IVR Server, you need to do the following:

- Translate escape characters to conform to XML parser rules. For example, use & instead of &. See “Escape Character Translation” on page 50.
- Ensure that you have a working socket connection.
- Format user data strings correctly. These strings represent key-value pairs that are required by the `UDataAddKD` and `UDataAddList` APIs. Formatting the strings correctly ensures that I-Library can create the correct XML message to send to IVR Server. For example, `ilSRqUDataAddKDList(0, 1,"*employee*smith*location*texas*")` contains a properly formatted user data string.
- Provide consistent call ID so that IVR Server can track call progress.

For more information about API processing see, “Processing API Requests” on page 46.

Diagnostics

In order to provide diagnostic information when customers perceive problems during call processing, configure your environment as follows:

- Create and update log files using configuration data supplied by the user.
- Provide sufficient diagnostic data in the logs to track individual calls and requests, socket traffic, error conditions, and date and time.
- Prevent sensitive data, such as user information, from appearing in log files; by configuring logging parameters appropriately.
- Prevent repetitive data, such as failed login events when the IVR Server is in standby mode, from appearing in the log files by configuring logging parameters.

Genesys

Chapter

2 Code Example One: Hello IVR World

This chapter introduces the I-Library functions that implement the most essential functionality of an IVR driver. This chapter has these sections:

The example code in this chapter is not tested. Its purpose is to show the order in which to make calls to the I-Library functions and to provide a structure for discussion. Treat it as pseudocode that looks a lot like C.

The most important feature of the I-Library is its use of requests and replies. Generally, your driver uses a request function to take an action and follows that with a call to a reply function to get related data.

Overview

This chapter has two major sections, one that presents partial sample code for a very small IVR driver and one that discusses the request/reply communications convention a driver uses to communicate with the I-Library.

An IVR driver must first initialize the I-Library, and then make contact with an IVR Server. After these tasks are done, the driver can perform other tasks.

For each call, the driver should notify an IVR Server of the beginning of the call, then perform necessary tasks, then notify the IVR Server that the call has ended.

Tasks that are related to a call include transferring and conferencing/merging as well as working with user data, routing strategies, statistics, and more.

Many of the IVR API functions that perform tasks are implemented as requests that allow the IVR Server to do the task and prepare a reply. For each request,

the I-Library creates a request ID. The driver typically should follow a request with a call to a `GetReply` function that retrieves data pertinent to the request ID.

A Simple Call Examined

The following code examples begin by revealing some of the constants and types defined in the supplied header file, `interface.h`. This section then presents two functions: `Start()` and `MakeSimpleCall()`.

The (untested) code examples in this section could be integrated with some `main()` function that brings in data. The examples are presented below in three subsections:

- Header File Data
- The `Start()` Function
- The `MakeSimpleCall()` Function

These sections provide a structure for the discussion by isolating related functionality.

Header File Data

The IVR SDK ships a header file, `interface.h`, that has definitions for various common types, such as `CPSTR` (pointer to a constant string), `ULONG` (unsigned long), and the IVR SDK–specific constants, types, and function declarations. (In the `interface.h` header file is a set of DOxygen comments that generate the HTML API reference pages.)

The header file features (commented out) in this section are just those that are needed for the sample `Start()` and `MakeSimpleCall()` functions presented in this chapter.

Code Fragment

```
#include "interface.h"
/* from the header file:
   typedef const char         *CPSTR;
   typedef unsigned long      ULONG;

   typedef ULONG           ilPORT;
   typedef ULONG           ilRQ;
   #define ilRQ_ERR        0
   #define ilRQ_ANY        0
   typedef long            ilRET;
   #define ilRET_OK        0L
   #define ilRET_ERROR     -1L
```

```
   // Inform IVR-Server of start/end of call processing:
   ilRQ    ilSRqNoteCallStart(ilRQ RqID,ilPORT Port,
           CPSTR psCallID,CPSTR psDNIS,CPSTR psANI,
           CPSTR psTagCDT); // last 4 args optional
ilRQ    ilSRqNoteCallEnd(ilRQ RqID,ilPORT Port);
*/

#define MAX_PACKET_SIZE 16384
char szResp[MAX_PACKET_SIZE]; /* global string buffer */
typedef enum
{ ilPT_CHAN_NUM, ilPT_DN } ilPORT_TYPE;
int Start(void);
void MakeSimpleCall(ilPort);
                        // called from Start()
```

Notice the definition of the `szResp[]` string buffer. The purpose of this global buffer is to store reply data, as discussed in the section “Requests and Replies” on page 27 later in this chapter.

The Start() Function

The `Start()` function isolates initiation and connection issues. Initiation sets the I-Library for communication with your IVR driver. Connection directs the I-Library to connect to a specified IVR Server.

Code Fragment

```
/* ... Here we go... -------------------------------*/
int Start(void)
{
   static char*    pIServerHost="EnterpriseIS";
   static char*    pIVR="me"; /* name of this IVR
                              in the Configuration Layer*/
   static char*    pIVRversion="3.51"; /* driver version */
   ilPort          nPort=110; /* must match config data */
   ULONG           ulTimeoutMS=5000 /* in milliseconds */
   ULONG           ulNetWatchTime = 0L;

      if ( ! ilInitiate(pIVR)
   {
      puts("ERROR! Cannot initialize");
      return(-2);
   }

   if (! ilSetVersion(pIVRversion))
   {
      puts("Error! Cannot set version");
      return(-1);
   }
```

```
   printf("Connecting to port %d\n",nPort);
   if( ! ilConnectionOpen(pIServerHost,nPort,ulTimeoutMS))
   {
      printf("ERROR! Cannot open connection to server!\n");
      return(-3);
   }

   MakeSimpleCall(nPort); /* <--- this makes a call */
   printf("Done. Goodbye!\n");
   return 0;
} /* end of Start() */
```

Initiation

The `Start()` function must call the `ilInitiate()` function and pass in the name of the IVR (as specified in the Configuration Layer) to initialize the I-Library for its use.

The IVR Server maintains a list of the channels and corresponding DNs for every IVR. For the IVR Server to properly track the requests it receives, it must associate a given connection with a set of DNs or channels. It does this by using the name of the IVR set in the Configuration Layer as the name of the connection. In this way, all messages from a given network connection through the IVR Server are automatically associated with the particular IVR driver and the IVR system that driver represents.

The IVR Server uses the IVR name to reference other configuration information as well.

The `Start()` function calls the `ilSetVersion()` function and passes its version number to the I-Library.

Opening the IVR Server Connection

The `Start()` function calls the `ilConnectionOpen()` function to direct the I-Library to open a connection to the IVR Server (this is not done automatically). The I-Library passes the IVR name to the IVR Server. Upon connection, the IVR Server sends a list of configuration parameters (from the `IVR` object's `Annex` tab in the Configuration Layer) to the I-Library. The I-Library transparently handles these setup parameters with no action required by the IVR driver. (The user can change these configuration parameters later, though any changes do not take effect until driver restart.) For example, this technique controls log file options. The connection to the IVR Server is automatically closed when the driver exits from the I-Libary.

Set Reply Latency

The `ilSetTimeout()` function takes a single unsigned long argument that specifies a time limit in milliseconds. The purpose is to allow the `ilGetReply()` and `ilGetRequest()` functions one or more cycles, balancing latency against improved chances to get a reply.

An argument that has a value of zero specifies exactly one complete cycle for the reply function. This value of zero means that the reply function decides immediately if the requested data is available, there is no delay to allow the data to be received during this reply cycle. Higher values allow repeated polling to improve chances of success.

The timeout value should be small because the `ilGetReply()` and `ilGetRequest()` functions, if no response has been received from IVR Server, will wait for the time specified before returning. During that time, the I-Library will continue to check for messages received from the IVR Server.

The MakeSimpleCall() Function

The `MakeSimpleCall()` function isolates the use of notification functions and introduces the convention of making a request and getting a reply.

T-Server Notification

The `MakeSimpleCall()` function uses the notification functions to notify the IVR Server about call processing by the IVR. The `ilSRqNoteCallStart()` function tells the IVR Server to signal the T-Server that call processing has started. The `ilSRqNoteCallEnd()` function signals that call processing has completed.

Notification Functions

```
ilRQ    ilSRqNoteCallStart(ilRQ RqID,ilPORT Port,
                           CPSTR psCallID,
                           CPSTR psDNIS,
                           CPSTR psANI,
                           CPSTR psTagCDT);

ilRQ    ilSRqNoteCallEnd(ilRQ RqID,ilPORT Port);
```

After the `NotifyCallStart`, the driver should ensure that the IVR Server is ready to process further telephony requests. To ensure that your driver is ready, take the following steps:

1. After issuing the `NotifyCallStart`, issue a `GetReply` call for this request.
2. Issue a `GetCallStatus` call on the line.
3. Verify that your application receives the value `eCallStatusEstablished`.

4. If it does not, continue to issue the `GetCallStatus` until it does, using a reasonable delay to allow the IVR Server to respond to the `NoteCallStart` request. If, after a reasonable period of time, the correct response has still not been received, issue a `NoteCallEnd`, and abandon the call since an error has occurred.

Making a Request

The notification functions are designed to request a reply. The notification functions return a request identifier to be used by a subsequent call to the `ilGetReply()` function. If that value is `ilRQ_ERR` or less, the function has failed, and a call to the `ilGetReply()` function will not be useful. See the section "Requests and Replies" on page 27 later in this chapter.

Getting a Reply

If the notification call succeeds, the IVR driver calls the `ilGetReply()` function, passing in the `ilRQ` value returned by the `ilSRqNoteCallStart()` function along with a pointer to the `szResp` string buffer and the size of that buffer. If the I-Library returns an `ilRET` value less than `ilRET_OK`, then the IVR driver must handle the error. In that case the `MakeSimpleCall()` function retrieves an error message string from the `szResp[]` string buffer.

Get Reply Function

```
ilRET      ilGetReply(ilRQ RqID,PSTR psRep,int iRepLen);
```

The `MakeSimpleCall()` function notifies an IVR Server when a call begins and ends.

In the case of a virtual T-Server that does not work with a switch, the notification functions simply tell the virtual T-Server that a call has started.

Code Fragment

```
static void MakeSimpleCall(ilPort nPortA)
{
   ilRQ          ilRq;

   printf("Starting call on channel %d\n",nPortA);
   ilRq=ilSRqNoteCallStart(ilRQ_ANY,nPortA);
   if(ilRq==ilRQ_ERR)
   {
      printf("Error making request\n");
      return FALSE;
   }
```

```
   if(ilGetReply(ilRq,szResp,sizeof(szResp))<ilRET_OK)
   {
      printf("Error notifying I-Server of call start\n");
      printf("Error Message: %s\n",szResp);
      return;
   }
   printf("Call is started\n");

   sleep(2);                  /* <--- THE CALL */

   /* call complete */
   printf("Ending call\n");
   ilRq=ilSRqNoteCallEnd(ilRQ_ANY,nPortA);
   if(ilRq==ilRQ_ERR)
   {
      printf("Error making request\n");
      return FALSE;
   }
   if(ilGetReply(ilRq,szResp,sizeof(szResp))<ilRET_OK)
   {
      printf("Error notifying I-Server of call end\n");
      printf("Error Message: %s\n",szResp);
      return;
   }
   printf("Call is ended\n");
   return;
} /* end of MakeSimpleCall() */
```

Making a Call

A two-second sleep in the example code represents call activity. Normally at this spot your driver would call several telephony functions.

Terminating

The `MakeSimpleCall()` function calls the `ilSRqNoteCallEnd()` function to notify the IVR Server that call activity has ceased, tests the return value, and gets reply data stored in the `szResp` string buffer (which overwrites the previous data).

Requests and Replies

The primary communications between your driver and the I-Library is implemented through requests and replies. You make a call to a request function, capture its return, then pass that return value to the `ilGetReply()` function, which writes data into a string buffer your driver maintains.

Requests and replies from the IVR Server are asynchronous, in that most replies do not immediately follow the request—other requests and replies are mixed in the data stream. It is good practice to follow a request with a call to the `ilGetReply()` function.

Request Functions

The IVR API provides two main types of functions, those that perform an action and those that perform an action and also make a request for an asynchronous reply. The prefix in the name of a function identifies the group of functions to which it belongs.

Functions that begin with the letters *il* (without the letters *SRq*) are those that perform an action without requesting a reply. Functions of this group usually do not generate a network message (request) to the IVR Server. The `il` functions are used to control the behavior of the I-Library and return either a `BOOL` result or an `ilRET` value.

Functions that begin with the letters *ilSRq* are those that take action and make a request for a reply. A request function makes an order and then generates and sends a special network message (request) to the IVR Server.

Order Term

When an `ilSRq` function is called, the I-Library creates an order, saves it, and sends a request message to the IVR Server. Orders provide a placeholder for tracking asynchronous requests and replies.

An order links a request and its associated reply. When a reply comes to the I-Library from IVR Server, the I-Library searches for the matched order to complete the previously sent request.

The order only lasts for the duration of the call on the port where it was originally requested. When that call ends, the order is discarded.

The design of the I-Library is such that it removes order numbers and any associated information structures when the `ilGetReply()` succeeds and the information is returned to the IVR driver. If, for some reason, the `ilGetReply()` function fails or is not correctly called by the IVR driver, the order is still removed at the end of the call.

Request IDs and Ports

Request functions pass in a first argument that is an `ilRQ` value, a second argument that is an `ilPort` value, and possibly additional arguments, depending on the purpose of the function.

Every request function must have a unique `ilRQ` request number associated with it. Your driver may either specify a particular number in the function call or allow the I-Library to assign the number (recommended).

If your driver specifies a particular `ilRQ` value, the driver must manage its validity and prevent contention with any `ilRQ` values the I-Library creates.

Note: The `ilRQ` value must be in the range `1` to `FFFFFE`, hexadecimal.

If the request function succeeds, it will return the `ilRQ` value you passed in.

If you pass in `ilRQ_ANY` as the first argument in a request function, the I-Library will create and return a unique `ilRQ` request ID and manage contention possibilities. (Because this number is a long integer, sequential requests have little risk of reusing a given number while the previously matched request number is still active.)

Note: While a request ID might be reused after the reply to the original request with that ID has been received, it is not valid to have two outstanding requests for the same call using the same request ID.

The `ilPort` value specifies a particular call associated with the port. The IVR driver must specify the IVR port number (as configured for the IVR in the IVR section of Resources in the Configuration Layer) to assure that the IVR Server can correlate the requests and responses for a particular call.

If a driver specifies a port that is not configured for that named IVR connection, then the IVR Server returns an error for all requests made with that port number.

Request Return Values

If a request function succeeds, it returns an `ilRQ` value that is a request ID. This request ID should be used in a subsequent call to the `ilGetReply()` function. Generally request functions should pass in an `ilRQ_ANY` and capture the return to pass in with the next `ilGetReply()` function call.

If a request function fails, it returns an `ilRQ_ERR` value. Your driver should test for this return value and handle any errors appropriately, according to the particular request function.

The ilGetReply() Function

Immediately after calling a request function, your driver should call the `ilGetReply()` function to retrieve response information for the request.

Input Arguments

The `ilGetReply()` function takes three arguments. The first is a request ID for an existing order, the second is a pointer to a string buffer that your driver

maintains for storing replies, and the third is the size of the string buffer in bytes.

The request ID is a return value from a request function.

The `ilGetReply()` function writes a string to your reply buffer, depending on the nature of the request that matches the request ID.

Return Values

The `ilGetReply()` function returns an `ilRET` value. If the return value is `ilRET_OK` or greater, then the `ilGetReply()` function has succeeded. If the return value is less than `ilRET_OK`, the `ilGetReply()` has failed. You may want to set up a switch statement to test the possible causes of errors, which are defined in the `interface.h` header file.

Code Fragment

```
#define ilRET_OK                    0L
   /*    or any positive value */
#define ilRET_ERROR                 -1L
#define ilRET_LIB_NOTREADY          -3L
#define ilRET_CONN_CLOSED           -5L
#define ilRET_BAD_ARGS              -7L
#define ilRET_FUNC_UNSUPPORTED      -9L
#define ilRET_TIMEOUT               -11L
   /*    Order still be in progress */
#define ilRET_REQ_EXPIRED           -12L
   /*    There is no request with specified number
         or it has been expired */
#define ilRET_NO_REQUESTS           -13L
   /*    There is no request in the queue for processing.
         (Generated by ilGetReplyAny()) */
#define ilRET_BAD_CONN_NAME         -15L
   /*    The connection name is bad */
#define ilRET_REQ_FAILURE      -1000L
   /*    Reply from requested service contains failure code */
```

If the failure is `ilRET_TIMEOUT`, using the same request ID, you can attempt the `ilGetReply()` function again. The timeout indicates that no reply has been received from the IVR Server. It would be appropriate to limit the amount of time that is allowed to wait for a response in case an error has occurred and no response will be returned.

Chapter

3 Code Examples: Basic Functionality

This chapter introduces IVR Library (I-Library) functions for version information, telephony, and handling user data. This chapter has these sections:

On the Genesys Documentation Library CD you can find a file named `IVRexample.c` that exercises many of the I-Library functions. In Chapter 5 of this document, you can find a digest of the API functions. In the documentation directory on the IVR SDK product CD you can find the most extensive documentation for the API in HTML format.

Overview

This chapter introduces four common functionalities that drivers implement:

- Getting Version Information: For informational purposes only, your driver can identify that it has connected to a compatible I-Library and IVR Server. Use the version functions to do this.
- Process Telephony Functionality: For IVRs in behind mode only, use the telephony call-processing functions on active calls.
- Manipulate User Data: The I-Library has a set of functions that enable your driver to manage user data.
- Initiate Routing: The I-Library allows you to initiate routing-sequence logic.

Get Version Information

The IVR API provides functions for working with given versions of the I-Library, the IVR Server, and your IVR driver.

Use the `ilSetVersion()` function to pass a string representation of your driver's version to the I-Library, as shown in the `Start()` function in "The Start() Function" on page 23. The ability to get the version of this IVR driver from the I-Library is pro forma because this IVR driver should be able to access its own information.

The `ilGetVersion()` function takes no arguments and returns a string representation of the I-Library version.

The `ilSRqVersion()` function directs the I-Library to create a reply string that represents the version of one of three components: this IVR driver, the current I-Library, and the active IVR Server.

The `ilSRqVersion()` function takes three arguments, an `ilRQ` value, a port value, and a string that identifies the component for which a version is wanted. If the third argument is a null pointer or a single blank space, the `ilSRqVersion` assumes a default request for the version of the current I-Library.

To get the data, the driver must subsequently call the `ilGetReply()` function and then access the driver's reply buffer.

Code Fragment

```
ilRq=ilSRqVersion(ilRQ_ANY,nPortNULL,NULL);
/* The third argument is seen as a null pointer to
   a string, which indicates the default, the version
   of the I-library. */

if(ilRq==ilRQ_ERR)
{
   printf("Error making request for driver version\n");
   return;
}
if(ilGetReply(ilRq,szResp,sizeof(szResp))<ilRET_OK)
{
   printf(
      "Error in getting version of library\n");
   return;
}
printf("Library Version: %s\n",szResp);

ilRq=ilSRqVersion(ilRQ_ANY,nPortNULL,pIVR);
   /*<-- pIVR is the IVR name, so a
         successful call will generate a reply that
         specifies the version of this driver. */
```

```
if(ilGetReply(ilRq,szResp,sizeof(szResp))<ilRET_OK)
{ /* handle error */ }
printf("Driver Version: %s\n",szResp);

ilRq=ilSRqVersion(ilRQ_ANY,nPortNULL,"I-Server");
   /*<-- any string other than " " or the IVR name */
if(ilRq==ilRQ_ERR)
{ /* handle error */ }
if(ilGetReply(ilRq,szResp,sizeof(szResp))<ilRET_OK)
{ /* handle error */ }
printf("I-Server Version: %s\n",szResp);
```

Telephony

The I-Library provides several telephony functions, including functions for transferring calls and performing merge operations. These functions apply only to IVRs deployed in behind mode.

Before you use the telephony functions, you must initiate the I-Library and make a connection to the IVR Server, as shown in Chapter 2, “Code Example One: Hello IVR World,” on page 21.

The following simple example shows the use of the `ilSRqCallInit()` and the `ilSRqCallComplete()` functions, which begin and terminate a call to a particular DN specified by the `psDstDN` string pointer. While the call is in progress, your driver may perform other call-handling activities.

Code Fragment

```
ilRq = ilSRqCallInit(ilRQ_ANY,Port,psDstDN);
if (ilRq == ilRQ_ERR)
{ ; } /* handle the error */
if (ilGetReply(ilRq, Port,    szResp,
                sizeof(szResp))<ilRET_OK)
{ ; }   /* handle the error */

/* perform other call handling activity */

ilRq = ilSRqCallComplete(ilRQ_ANY,Port);
if (ilRQ == ilRQ_ERR)
{ ; }   /* handle the error */
/* call ilGetReply() as usual */
```

The `IVRexample.c` file exercises most of the I-Library’s telephony functions.

User Data

User data is stored as a single ASCII string in the form of:

```
DelimiterKeyDelimiterDataDelimiter...
```

where the delimiter is a single character that is defined by virtue of the fact that it is the first character in the string, keys occupy the even-numbered fields (assuming zero-based fields), and data occupy the odd-numbered fields.

Here is an example of a set of keys and data:

```
%inquiry%problem%agent%nancy drew%status%researching this
```

where the field delimiter is the % character. The key, inquiry, has a companion value, problem, whose meaning is that the nature of an inquiry is a problem; the agent is nancy drew; and the current status is that she is researching this.

There are several ilSRq request functions that enable you to work with key-data (key-value) pairs (also known as user data). You can add or delete a key-data pair, add a list of key-data pairs, or delete all key-data pairs for a particular call interaction. See the IVRexample.c file for extensive examples.

In the following code snippet, the ilSRqUDataAddKD() function creates a new key, Name, with a value of Dotty. The subsequent code tests to see if the function failed, and if so, prints an error indication and returns it. If the add function succeeds, then the driver calls the get reply function and, if that fails, captures an error message in the szResp string buffer.

Code Fragment AddKD

```
ilRq=ilSRqUDataAddKD(ilRQ_ANY,Port,"Name","Dotty");
if(ilRq==ilRQ_ERR)
{
   printf("Error making request\n");
   return;
}
if(ilGetReply(ilRq,szResp,sizeof(szResp))<ilRET_OK)
{
   printf("Error in attaching data\n");
   printf("Error Message: %s\n",szResp);
   return;
}
```

To get a value for a known key, use the ilSRqUDataGetKD() function and then use the ilGetReply() function to direct the I-Library to put the value in the szResp string buffer.

Code Fragment GetKD

```
ilRq=ilSRqUDataGetKD(ilRQ_ANY,Port,"Name");
if(ilRq==ilRQ_ERR)
{ /* handle error */ }
if(ilGetReply(ilRq,szResp,sizeof(szResp))<ilRET_OK)
{ /* handle error */ }
printf("Name: %s\n",FindRespStart(szResp));
```

You can delete a key-data pair, delete all key-data pairs, or add a set of key-data pairs.

Code Fragment DelKD

```
ilRq=ilSRqUDataDelKD(ilRQ_ANY,Port,"Name");

ilRq = ilSRqUDataDelAll(ilRQ_ANY,Port);

pcTheList = "&Name&Lion&Address&456 Yellow Brick Rd."
ilRq=ilSRqUDataAddList(RqID_ANY,Port,pcTheList);
```

To delete a key-data pair, use the `ilSRqUDataDelKD()` function and pass in `ilRQ_ANY`, the port for the call, and the key (a string pointer). (Capture error messages in the `szResp` buffer by calling the `ilGetReply()` function.)

To delete the entire set of key-data pairs for a particular call, use the `ilSRqUDataDelAll()` function.

To add a set of key-data pairs, first build a single string with your choice of a field delimiter character, and then use the `ilSRqDataAddList()` function and pass in a pointer to that string.

Initiate Routing

The I-Library offers functionality for you to implement routing-sequence logic. Before you use the routing functionality, you must initiate the I-Library and make a connection to the IVR Server, as shown in Chapter 2, “Code Example One: Hello IVR World,” on page 21.

The following simple code fragment outlines the use of the `ilSRqRouteStart()` function, which initiates the routing of a call to a particular route point DN specified by the DN number, as in:

```
ilRQRouteStart = ilSRqRouteStart(ilRQ_ANY,Port,"7000");
```

Code Fragment RouteStart

```
/* For information on each of the following, see the interface.h:

Port              = type ilPORT with the port of IVR channel
ilRQRouteStart    = type ilRQ - returned Request ID for Route Start
```

```
psRouteStartRep    = type PSTR - pointer to buffer - preallocated - in which GetReply
   returns its result
iRepLen            = type int - length of above buffer
ilGetReplyRet      = type ilRet - returned from GetReply indicating result of request
ilRQGetRequest     = type ilRQ - returned Request ID for GetRequest - used in SendReply
ilSendReplyRet     = type ilRet - returned from SendReply indicating result of request
bResult            = type BOOL - set true if treatment was success - else false
psReply            = type CPSTR - pointer to buffer with result of treatment - if return
   is required

ilRQ_ANY - generate the request ID

nlrepeat           = a number if type int to indicate strategy still active
*/

ilRQRouteStart = ilSRqRouteStart(ilRQ_ANY,Port,"7000");
                                        /* route sequence start on route dn -
                                        note that the port variable, in this
                                        case "7000," must contain the actual
                                        port value */

if (ilRQRouteStart > 0)                 /* make sure Route Start worked */
{
   nlrepeat = 1;
   while(nlrepeat == 1)
   {
      ilGetReplyRet = ilGetReply(ilRQRouteStart,psRouteStartRep,iRepLen);
                                        /* check reply for Route Start */

      if(ilGetReplyRet == ilRET_TIMEOUT)
                                        /* timeout means that the routing
                                        strategy is still active */
      {
         ilRQGetRequest = ilGetRequest(Port,psRep,iRepLen);
                                        /* retrieve the next treatment - if one
                                        exists - treatment details in the return
                                        buffer */

         if(ilRQGetRequest>0)           /* make sure Get Request worked before*/
         {                              /*processing the returned treatment
                                        processing by application to apply
                                        treatment */
            ilSendReplyRet = ilSendReply(ilRQGetRequest,bResult, psReply);
                                        /* send treatment result to URS */
         }
      }
```

```
      else    nlrepeat = 0;             /* The GetReply function returned
                                        something other than timing out - this
                                        implies the strategy has ended - and a
                                        RouteResponse has been processed by the
                                        library - the route destination - if it
                                        exists in the RouteResponse - will be
                                        returned in psRouteStartRep */
   }
} else                                  /* RouteStart failed */
```

Genesys

Chapter

4 Extended Functionality

This chapter introduces extended functionality, including, logging, KeepAlive processing, outbound dialing, load sharing, and routing. It has these sections:

Configuration Data

I-Library uses configuration data to determine how to direct processing. For example, there are options for interfacing with IVR Servers and for modifying log content. I-Library has several levels of configuration.

- Internal defaults—These are used if no options are configured.
- `ivrlibrary.ini`—At startup, if this file exists in the directory where I-Library is executing, any options configured here take effect and over-ride the internal defaults.
- Configuration Layer options—When the options configured in the Configuration Layer are received, they take effect and over-ride the `.ini` file options.

I-Library retrieves the configuration options through the Configuration Server. At startup, I-Library attempts to connect to Configuration Server. If I-Library cannot connect, it exits. If a disconnection occurs after successfully connecting to Configuration Server, I-Library will attempt to reconnect to Configuration Server. Until the connection to configuration server is restored, I-Library will continue to process telephony APIs although dynamic configuration updates will not be available.

Table 1 explains which options you can set, where you can set them, and what their default values are. The "Driver Application" column applies only to "7.0 Mode." The "Data Transport" column applies only to "6.5 Mode."

Table 1: Configuration Options

Configuration Item	Driver Application	IVR Object DataTransport Section	ivrlibrary.ini	Default Value
ivr_server_interface	section			
load_sharing_servers	yes	yes		n/a
time_recon_is	yes	yes	yes	2000ms
socket_activity_timer	yes	yes		20000ms
load_sharing_iservers_ client_ports	yes			n/a
load_sharing_iservers_ client_hosts	yes			false
cfg-server-response			yes	false
getreply_with_location	yes			false
iserver_mode_ hotstandby	yes			false
compat65	yes	yes		
log	section			
verbose	yes			all
all	yes			n/a
log_content	section			
log_print_level	yes	yes	yes	xml
log_file_name		yes	yes	con

Table 1: Configuration Options (Continued)

Configuration Item	Driver Application	IVR Object DataTransport Section	ivrlibrary.ini	Default Value
log_file_backup_amount		yes		0
log_file_size		yes		0
log_print_date	yes	yes		yes
log_print_hb	yes	yes		no
log_print_name	yes	yes		yes
log_print_recv	yes	yes		yes
log_print_send	yes	yes		yes
log_print_time	yes	yes		yes
log_print_time_ms	yes	yes		yes
log_print_timeouts	yes	yes		no
log_print_udata	yes	yes		no
log_print_login_requests	yes	yes		yes
log_print_driver_selector	yes	yes		0
dtd_version			yes	4.0

Logging

I-Library Log Files

Under normal operation, I-Library receives its log file name from the Configuration Layer; therefore, no log file will be created until that information is received. However, a log file name and the level of logging can be configured in the `ivrlibrary.ini` file. In this way, in the event of startup problems, the `.ini` file can provide information about processing during that startup.

Note: I-Library log files are subject to change at any time and should not be considered a defined interface.

If an invalid path is defined when configuring logging, I-Library will detect it and stop execution.

The ivr library.ini File

Use the `ivrlibrary.ini` file to provide initial values for certain I-Library options. However, these values can be overridden if the Configuration Layer provides different values after its connection to I-Library. Table 1 on page 40 shows which options can be used in the `.ini` file. The format of the data in this file is important. To view the proper format see the sample file that was sent with I-Library.

Setting Log Levels

There are three levels of logging that will be used in most environments. Under normal conditions, it is recommended that you set the level of logging to `xml`. This will provide all messages flowing into and out of I-Library, including the XML messages being sent between I-Library and IVR Server. This is a minimum level of logging that you need in order to be able to diagnose any problems.

You can make the level of logging more detailed by using either `debug` or `detail` level. The `debug` level of logging will include all of the `xml`-level data, and it will also provide information about various I-Library internal tables and the basic flow.

The `detail` level of logging will include all of the `debug`-level data, and it will as provide more detail about the logic flow and the status of all sockets. This dramatically increases the amount of data that is logged, because messages are logged each time I-Library checks the socket to IVR Server to determine whether there are any messages waiting to be read and processed. I-Library checks the socket every 10 milliseconds (ms). Table 2 describes the available log levels.

Table 2: Log Levels

Log Level	Description
none	I-Library will use what was configured in the verbose option of the Log section.
flow	I-Library will log all messages flowing to and from the driver application, and to and from the IVR Server.
xml	I-Library will log all messages defined by `flow`, as well as all XML messages flowing to and from IVR Server.

Table 2: Log Levels (Continued)

Log Level	Description
debug	I-Library will log all messages defined by `xml`, as well as information about internal I-Library tables and basic program logic.
detail	I-Library will log all messages defined by `debug`, as well as information about socket activity and detailed program logic.

DTD Versions

The Document Type Definition (DTD) version is used in the XML message to define which versions of messages the sender of the XML is using. The DTD version will change if an existing message format is changed. There are three versions of the DTD defined for the communication between IVR Server and I-Library. Table 3 describes which release levels of I-Library and IVR Server support each DTD version. A login to IVR Server is first attempted using DTD version 4.0, if that is rejected, an attempt is made with version 3.0, and then finally with 2.0. Once IVR Library selects a DTD version, if you change the DTD version you must stop and re-start the IVR Library for those changes to take effect. Load Sharing IVR Servers must be at the same DTD version.

Table 3: Supported DTD Versions

I-Library /IVR Server Release	DTD 2.0	DTD 3.0	DTD 4.0
6.5	Yes	No	No
7.0	Yes	Yes	No
7.1	Yes	Yes	No
7.2	Yes	Yes	Yes
7.5	Yes	Yes	Yes
8.0	Yes	Yes	Yes
8.1	Yes	Yes	Yes

If the DTD version is configured as 2.0 in the `ivrlibrary.ini` file, I-Library will first attempt to connect using 2.0. This should be successful in any environment, because both the 6.5 and 7.x releases of IVR Server can use DTD

version 2.0. Nevertheless, in an environment that uses only 7.x IVR Servers, there is no reason to use 2.0.

Table 4 describes the new features that are provided in each DTD version after 2.0.

Table 4: New Features of DTD

DTD Version	Feature
3.0	I-Library can retrieve First Home Location by using `GetCallInfo`.
4.0	I-Library can control agent login, and so on.

Call State Model

A call state model is kept for each call that is tracked by the port it is on. It is used to determine whether a request will succeed if it is sent to the IVR Server. If the request would fail, an error is returned to the application, and the request is not sent to the IVR Server. In general, the call state is set based on messages received from the IVR Server—for example, a `CallStatus` message with `Call Established` will set the call state to established. The call state is based on the definitions in the *IVR SDK 8.1 XML Developer's Guide*.

In addition, route requests are tracked to ensure that only one route request on a port is active at any one time. If a second route request is received before the first is done, the second one will be rejected. The same is true for `CDT_Init` processing.

Table 5 describes the defined call states for I-Library.

Table 5: Call States for I-Library

Call State Enum	Description
eCallStatusUnknown	The call state is unknown. This usually indicates there is no call on the port.
eCallStatusDialingMakeCall	A make call is in progress, but it has not yet been answered.
eCallStatusBusyMakeCall	A make call was made to a busy number.
eCallStatusRinging	A call has been made, and the called party's phone is ringing.
eCallStatusHeld	A call is on hold.
eCallStatusBusy	A call was made to a busy number.

Table 5: Call States for I-Library (Continued)

Call State Enum	Description
eCallStatusDialing	A call is being dialed.
eCallStatusEstablished	A call has been connected to the called party.
eCallStatusRetrieved	A previous consult call has been retrieved.
eCallStatusConfPartyDel	A party that was brought into a conference call has been deleted from the call.
eCallStatusConfPartyAdd	A party has been added to the call.
eCallStatusXferComplete	A call has been transferred.
eCallStatusReleased	A call has ended.
eCallStatusNoChange	Internal state used by I-Library. This would never be returned to the application.

KeepAlive Processing

As described in the GDI specification, I-Library provides a KeepAlive methodology. A configurable parameter, `socket_activity_timer`, is provided in the Configuration Layer to control the I-Library KeepAlive processing

The default value for `socket_activity_timer` is 20 (seconds).

A value of `0` (zero) disables KeepAlive processing.

The range of valid values is =>`1000` (ms).

If the value is greater than `0`, KeepAlive processing is enabled. This means that I-Library will use the `socket_activity_timer` value to determine whether a `KeepAlive` message should be sent to IVR Server. I-Library will sleep for the value of `socket_activity_timer`, and then determine whether any messages have been received from IVR Server.

- If at least one message has been received at the time of the check, I-Library will sleep again for the same amount of time.
- If no messages have been received, I-Library will send a `KeepAlive` message to IVR Server. I-Library will sleep again for the same amount of time, and then check again whether at least one message has been received from IVR Server. If the socket is connected, IVR Server will respond to this message with a `KeepAlive` response message, and I-Library will know that the socket is connected.

If I-Library has sent a `KeepAlive` message to IVR Server and, after sleeping for the value of `socket_activity_timer`, there is still no message, I-Library will sleep one more time, and then check again. This is to meet the criteria in the

GDI specifications. Do not assume that the socket is disconnected until no messages have been received after three times the configured amount of time. If, after sleeping this third time, no messages have been received from IVR Server, I-Library assumes that the socket is disconnected and, if necessary, starts the reconnect thread.

Processing API Requests

The API request is received from an application, which may be a Genesys driver. The APIs that begin with `il` receive an immediate reply. Those which begin with `ilSRq` must wait for a response from IVR Server before they receive a reply. The one exception to this is the `ilSRqGetCallInfo` request, which in certain circumstances, can respond without waiting for a response from IVR Server.

API processing proceeds as follows:

1. The first step in processing any API is to ensure that the library has been initialized. The library is initialized by the API call `ilInitiate`, which specifies which IVR resource will be used during the processing of calls.

   The only API that can be sent before `ilInitiate` is `ilGetVersion`.

2. The next step is to make sure that there is at least one IVR Server logged in to, ready to receive requests and respond to them.

   There are several APIs that may be issued before this —for example, `ilGetTimeout`, `ilLocalPrn`, `ilPrnError`, `ilSetLogHeader`, `ilSetTimeout`, and `ilSetVersionNumber`, which are processed without checking for IVR Server connectivity.

3. Next, for some APIs, the call state for the port on which the request is made is checked, to determine whether the request will succeed. If not, the call is rejected.

   Each API with string input parameters is checked, to ensure that there are no null strings. If a null string is found, for most APIs, an error condition is returned, and the API is not processed. There are some APIs, such as `ilInitiate`, that will exit with a null string pointer, because continuing does not make any sense. If the API exits, a log file named `I-Library_exit.log` is created in the directory where the driver is running. The file contains information pertaining to the reason for the exit.

Chapter 5, “IVR API at a Glance,” on page 59 contains detailed information about the APIs that are supported, and the XML message (if any) that they send to IVR Server.

Note: The driver application must insure that API calls are synchronized; that is, after an API is issued, no other is issued until the previous call returns. If this sequence is not followed, the results are indeterminate. This warning is for those who may have a multi-threaded application which is making unsynchronized calls to the SDK.

Processing Response Messages

The response message is usually a result of a previous request that an application made to I-Library. When a response message is received, I-Library must determine which request was responsible for it, and what data must be saved in anticipation of an `ilGetReply` looking for the data that it contains or represents.

Not all response messages result in a request ID being updated. The `CallStatus` message, for instance, indicates that a call state has changed. It is used to update the call state on the port on which the message is reporting. There are also `CallStatus` messages that indicate that an error has occurred. These will be described in the section dealing with IVR Server communications (see page 56).

For all other response messages, an attempt will be made to update the request ID that was created during processing of the API that precipitated this response message. I-Library maintains a request ID table, containing an entry for each API that it receives. This is the repository for the data that the response message contains. Some response messages contain the request ID of the API request that precipitated it. If the request ID is present in the response message, that request ID is updated with the information in the response message.

Response message processing proceeds as follows:

1. The first step in the processing of a response message is to determine its message type. This is done based on the Document Type Definition (DTD), which describes the XML format being used. The message type is always included in the message. Obviously, if a new type of message is developed, it must be added to this list, and new methods must be developed to extract its information.
2. After the message type is known, for those response messages which do not contain request IDs, the message must be linked to the request that precipitated it. Each entry in the request ID table contains the type of XML request message generated by the API. Because each response message—

with the exception of a `CallStatus` response message—is generated by only one type of request, linking the message to the request involves locating the request ID that contains the appropriate request message.

Finding the Appropriate Request Message

Remember that there might be multiple outstanding requests for each call, and that these requests might have generated the same request message. Therefore, the appropriate request message for this response message is the earliest one in the table, because responses will usually be returned in the order in which they were requested.

1. In order to find the earliest request containing the appropriate request message, the XML call ID is extracted from the response message. It contains a numeric call ID that was created and sent in the request message. I-Library maintains a table of active numeric call IDs. Each entry contains the port on which the call was received. The numeric call ID, therefore, is used to find the port with which this message is associated. If the call on the port with which this call ID is associated has ended, the call ID entry is reset. This becomes important when a response message for a call is received after a call has ended. The reset entry indicates that the response message is no longer of use, and that it can therefore be discarded.
2. After the port number is found, it is used to obtain the information in the I-Library port table that has an entry for each active port. Included in each entry is the list of request IDs associated with the call on that port. I-Library can now search each request ID entry in the request ID table, to find the earliest request message associated with this response message. The appropriate information in this response message is then saved in the request ID table entry, and this entry is marked as having been updated. The information is retained until the call is ended.

Error Codes

I-Library issues error codes for API requests attempting to indicate whether or not the request succeeded. Table 6 on page 49 identifies each error code, its value, and its meaning.

Table 6: Error Codes

Error Code	Error Value	Error Meaning	Error Correction
ilRQ_ERR	0	This request ID value is returned on the API call if it failed. The API will never return a negative number and any positive number is a valid request ID. In order to determine why the API failed, either look in the log file or issue an ilGetLastError.	
ilRET_OK	0	The API request was accepted by I-Library, with no errors found.	N/A
ilRET_ERROR	-1	This is a general error that does not fit into any other categories—for example: • A program exception occurred. • A request was made, but the call state was incompatible. • A route request was received, but there was already a route in progress. • An ilLocalPrnSelector was received, but no 1s was configured in the Configuration Layer to enable a message to be printed.	Review the previous messages in the log file, and make corrections as appropriate.
ilRET_LIB_NOTREADY	-3	The API request was sent before the ilInitiate API.	Send ilInitiate as the first API.
ilRET_CONN_CLOSED	-5	The API request was sent when there were no IVR Servers that I-Library was logged in to.	Start an IVR Server that I-Library is configured to connect to.
ilRET_BAD_ARGS	-7	The API request had at least one bad argument.	Correct the argument that is in error.
ilRET_FUNC_UNSUPPORTED	-9	ilInitiate was issued, and I-Library is already initialized.	Do not send ilInitiate more that once.
ilRET_TIMEOUT	-11	The GetReply or GetRequest API request found no response from IVR Server for the specified request ID.	Resend the request, giving the IVR Server sufficient time to respond.

Table 6: Error Codes (Continued)

Error Code	Error Value	Error Meaning	Error Correction
`ilRET_REQ_EXPIRED`	-12	The `GetReply` or `GetRequest` API request, which specifies which request ID to respond to, did not find it in the list of request IDs. This is either: • An invalid request ID. • A request ID that has already been responded to in a previous request. • A request ID from a call that has ended.	Do not use the request ID any longer.
`ilRET_NO_REQUESTS`	-13	For a `GetReplyAny` API request, there is no request ID with response data from IVR Server.	Try again later.
`ilRET_BAD_CONN_NAME`	-15	Not used.	N/A
`ilRET_REQ_FAILURE`	-1000	Indicated that one of the following occurred: • For a response message, the XML format was incorrect. • A `CallError` response message was received for a previous API request. • For a `GetCallInfo` response, the type of data requested was unknown.	Check the log for more information about which problem has occurred.

Escape Character Translation

XML has certain special characters that are used in the parsing of messages. I-Library must translate these characters to escape characters before sending them in an XML message. Likewise, when an XML message is received from IVR Server, any escape characters in the message must be converted to their special characters before sending them back to the application. Table 7 defines the special characters and their escape equivalents.

Table 7: Escape Characters

Escape Character	Converted Value
<	&lt;
>	&gt;

Table 7: Escape Characters (Continued)

Escape Character	Converted Value
&	&
“	"
‘	'
\t	
\r
\n

Note: This translation to escape characters in messages being sent to IVR Server occurs only for user data.

Routing

I-Library will format a `GetReply` response for the various types of responses that can be received at the end of a route sequence. This section lists some of the types of route responses that I-Library expects, and the value that subsequent `GetReply` will return for each one. The value that `GetReply` returns is historical, and it is kept the same across releases in order to be consistent with what applications are expecting.

Normal Route

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE GctiMsg SYSTEM 'IServer.dtd'>
<GctiMsg>
   <CallId>IVR·1·2</CallId>
   <RouteResponse RouteType='Normal'>
      <Dest>7777</Dest>
      <ExtnsEx>
         <Node Name='CUSTOMER_ID' Type='Str' Val='Resources'/>
         <Node Name='DN' Type='Str' Val='7777'/>
         <Node Name='NVQ' Type='Int' Val='1'/>
         <Node Name='SWITCH' Type='Str' Val='Virtual Switch'/>
         <Node Name='TARGET' Type='Str' Val='7777_Virtual
Switch@.RP'/>
      </ExtnsEx>
   </RouteResponse>
</GctiMsg>
```

The `getReply` should return `7777@'Virtual Switch'` if configured to return the switch name

Default with No Destination Address or Nodes

This is a default route response XML message with no default destination.

```
<?xml version='1.0' encoding='iso-8859-1'?>
<!DOCTYPE GctiMsg SYSTEM 'IServer.dtd' >
<GctiMsg>
   <CallId> SIL·1000·2 </CallId>
   <RouteResponse RouteType='Default' />
</GctiMsg>}
```

The `GetReply` request should return `RouteType Default`.

Outbound Dialing

I-Library has the ability to handle outbound calls. I-Library will communicate with both IVR Server and the driver/application, to ensure that call progress messages are flowing correctly between them.

1. At startup, I-Library will determine whether the configuration parameter `dial_out_dns` is populated with values indicating that I-Library should register with IVR Server in order to receive information about outbound calls.

   If the configuration indicates that it should, I-Library will send a `DialOutRegistry` XML message to IVR Server. IVR Server will respond with a `DialOutRegistryResp` XML message, indicating whether or not the registration process has been successful.

2. After successful registration has occurred, IVR Server will send any outbound call requests to I-Library, using the `DialOut` XML message. I-Library will accumulate these outbound call requests, waiting for the driver/application to request them.

3. When the driver uses the `ilSRqGetDialOutData` API to request a call, I-Library will return a request ID for the next call on its queue. The driver will use this request ID in an `ilGetReply` API, to request the information about the call that it needs in order to process it.

4. When the driver has successfully initiated the call, it will inform I-Library by using the `ilSRqDialOutDataInit` API, indicating which port the call was made on. I-Library will respond to this request with a request ID, and it will send the call information to IVR Server, using the `DialOutInit` XML message. IVR Server will respond with a `CallStatus` XML message with a status of `Dialing`. The driver will issue an `ilGetReply` API to determine whether the call initiation was communicated successfully.

5. When the driver has determined that the call has been established, it will use the `ilCallEstablished` API to inform I-Library. I-Library will respond with a request ID, and it will send a `CallStatus` XML message to IVR Server, with a status of `Established`. IVR Server will respond with two `CallStaus` XML messages, one with status `Ringing`, and the next with status `Established`.
6. The driver will now repeat this cycle by issuing another `ilSRqGetDialOutData` API to get the next call on the queue.

Results of the Request

When the driver requests the next call in the I-Library queue, but there are none remaining, I-Library responds with a request ID of `0` (zero).

If the driver issues the `ilSRqDialOutDataInit` API, and I-Library sends the `DialOutInit` XML message to IVR Server, but it fails because the port that the driver used was invalid or the call timer expired, IVR Server will send a `CallStatus` XML message, with a `feature not supported` indication.

If a dialing error occurs when the driver attempts to make the call, the driver should send the `ilDialOutError` API to I-Library, indicating the type of error that occurred: `NotSupported`, `NoTrunks`, or `MiscError`.

If the call fails, the driver should send a `ilFailure` API, indicating the type of error that occurred: `Busy`, `NoAnswer`, or `ConnectFailed`.

Connections and Load Sharing

IVR Servers and Load Sharing

I-Library will attempt to communicate with the IVR Servers that the user has configured.

In “6.5 Mode,” the `ilConnectionOpen()` API call from the driver identifies the IVR Server to which a login request is sent. The login response might identify other IVR Servers to which a login request should be sent.

In “7.0 Mode,” the `ilConnectionOpenConfigServer()` API call causes I-Library to read the configuration data from the configuration server that is retrieving the list of load sharing IVR Servers. Attempts will be made to log in to all of these IVR Servers. See for more details regarding this mode.

In “6.5 Mode,” the primary IVR Server is the one that is configured in the Data Transport option of the IVR object. The other IVR Servers use the primary server's configuration. There is no difference in operation between the primary IVR Server and the others. Any configured IVR Servers can be started first, and the others, when started, will operate satisfactorily.

In "6.5 Mode," the `ilConnectionOpen()` API identifies a single socket. If this primary IVR Server cannot be logged into, the I-Library will continue to attempt to log in to it. After the timeout that the IVR Driver specifies, the I-Library will send back a return code of `false`. Most drivers will not attempt to send any further APIs at this point. The Data Transport section information would not be available, and therefore no other IVR Servers will be known about to log in to.

In "7.0 Mode," the configuration data identifies all of the IVR Servers to be logged in to. I-Library will attempt to log into all of them, and if none can be logged in to, it will return a return code of `false`.

If there are multiple IVR Servers configured, an algorithm is used to select an IVR Server to which to send each call. The algorithm is simply to divide the port that the call is on by the number of IVR Servers available to process the call, then the remainder is used to select which IVR Server to use for the call.

For example, if there is a call on port 7 and there are three IVR Servers available to process the call, the IVR Server that is in position 1 in the list of IVR Servers would be chosen to send the call to (7/3 yields a remainder of 1). This call would continue to be sent to this IVR Server until the call on this port ends. At that point, the algorithm would be used again to determine which IVR Server to send the new call on this port to. This must be done in case the list of IVR Servers changes.

If any socket becomes unavailable, either because it is down or because has been taken off the socket list, the call in progress on that socket will have any API requests rejected. When a new call comes in on that port, the algorithm for selecting the socket to use will select another available socket.

IVR Servers and High Availability (Hot Standby)

An alternative to load sharing the IVR Servers is to configure the IVR Servers in high availability Hot Standby mode. When configured in this mode, one of the Servers will be considered the primary and will process calls. The other server, the backup, will maintain the same state as the primary in order to be available to immediately take over if the primary fails.

I-Library, if configured to communicate to the IVR Servers in this mode, will login to each IVR Server and from then on will send API requests to both servers and will expect only one response. I-Library will consider the IVR Server which sent the last response to be the primary. If I-Library detects that a server is no longer communicating, it will attempt to reconnect to it and will continue to send API requests to the server that is communicating.

Connecting to IVR Server

After obtaining sufficient configuration data, I-Library will attempt to connect, via a socket, to the IVR Server specified in the defined configuration. As I-Library attempts to connect to IVR Server, it makes use of two particular

configuration options, `time_recon_is`, and the timeout specified in the `ilConnectionOpen()` or `ilConnectionOpenConfigServer()` function.

The timeout value is the maximum amount of time that I-Library will take before responding to the application regarding the connection to IVR Server.

- If I-Library is able to successfully log in to IVR Server, a positive or `true` response will be returned on the `ilConnectionOpen()` or `ilConnectionOpenConfigServer()` function as soon as the login is successful.
- If I-Library is unable to successfully log in to IVR Server within the amount of time specified in the timeout value, a negative or `false` response will be returned. I-Library will actually continue to attempt to log in to IVR Server after sending the negative response, and it allows the application to attempt another `ilConnectionOpen()` or `ilConnectionOpenConfigServer()` function call. Only the timeout value will be honored for subsequent calls.

The `time_recon_is` value is the amount of time that I-Library will wait between attempts to connect and log in to IVR Server. So, for example, if I-Library attempts to log in to IVR Server, and the attempt fails because the socket is not available because IVR Server is not running, I-Library will wait the amount of time specified by `time_recon_is` before making another attempt to log in to IVR Server.

It is recommended that you set the `time_recon_is` value to `2000` ms, and that the timeout specified on the `ilConnectionOpen()` or `ilConnectionOpenConfigServer()` function be `60000` ms—or greater, if the IVR Server has a significant number of DNs to register. Experience has shown that, with an IVR Server registering in the range of 1000 DNs, the timeout value needs to be greater than 60000 ms.

While I-Library is attempting to log in to IVR Server, I-Library will log progress messages into the configured log file.

The connection to I-Server may be made secure by configuration. Refer to the details in the *IVR Interface Option 8.1 IVR Server System Administrator's Guide*

Connection Problems

There are several conditions that can cause I-Library to be unable to connect to IVR Server.

- The most obvious is that the IVR Server is not running. I-Library will continue to attempt to log in to IVR Server until the process is stopped. This is the case whether a single IVR Server or multiple IVR Servers are configured. If multiple IVR Servers are configured, I-Library will treat them separately when attempting to log in to each one. If one IVR Server is

running, and I-Library successfully logs in to it, and the other IVR Server is not running, I-Library will continue to attempt to connect to the IVR Server that is not running.

- Network delays can cause problems when I-Library is attempting to log in to IVR Server. Experience has shown that if I-Library and IVR Server are in different physical locations, a simple ping can take as long as 300 to 400 ms. In this situation, it takes time for the socket to inform I-Library that it is available to write to, and therefore I-Library is designed to allow up to 1000 ms before determining that the socket is unusable, closing it, and attempting to create a new socket. If your network is this slow, you will be able to have I-Library log in to IVR Server; however, your call handling volumes might be substantially degraded.
- Network delays can also cause delays in the connection of the socket from I-Library and IVR Server resulting in an `in progress` messages from TCP/IP. I-Library is designed to attempt 10 retries on the socket before determining that the socket is unusable, closing it, and creating a new one. In this situation one retry is usually sufficient in order to have a successful connect.

Connecting to IVR Server After Startup

If IVR Server stops running after having been connected to I-Library, I-Library will recognize that IVR Server is no longer connected and take steps to reconnect to it. I-Library will attempt to connect and log into IVR Server using the same design as when it initially connected. The problems that are experienced, therefore, are the same as when initially connecting.

Handling IVR Server Disconnects

After I-Library and IVR Server are successfully communicating, communications between the two can be interrupted by network problems or by IVR Server exiting. When either of these happens, I-Library is designed to attempt to reconnect to IVR Server forever.

Processing Calls

Communications to IVR Server Is Interrupted

When I-Library detects that IVR Server is no longer communicating, it begins an analysis to determine which ports were using that IVR Server and, for each of those ports, whether there are any request IDs still in-progress—that is, no response has yet been received from IVR Server. Each of those request IDs is deleted from the request ID table. The effect of this is to return `ilRET_REQ_EXPIRED` to an `ilGetReply` that is issued for that request ID. Because the IVR Server is no longer available to process request messages, any APIs

issued for the ports that were using that IVR Server will be responded to with `ilRQ_ERR`, and the last error will be set to `ilRET_CONN_CLOSED`. If a new call is started on this port, and there are other IVR Servers available, the call will be switched to another available IVR Server.

Communications to IVR Server Is Restored

If I-Library is able to reconnect to the IVR Server, any APIs using the call ID that was being used when communications with the IVR Server were disrupted will still be sent to the IVR Server. At this point, two possible responses from IVR Server can occur:

- If the communications disruption was due to a network problem, and IVR Server still has the call model available for the calls that were in progress when the miscommunications occurred, IVR Server will respond to the request, and I-Library will handle it.
- If the communications disruption was due to IVR Server exiting, IVR Server will no longer have the call model available, and it will respond to the request with a `CallError` message, with an explanation of `NoSuchCall`. When I-Library processes this message, the state of the call on the port on which this request was made will be set to `eCallStatusReleased`. The effect of this is that any APIs except `ilSRqGetCallInfo` and `ilSRqNotifyCallEnd` will be returned with `ilRQ_ERR`, and the last error will be set to `ilRET_ERROR`. Both the `ilSRqGetCallInfo` and `ilSRqNotifyCallEnd` APIs will be given valid request IDs and processed appropriately, because they are always allowed to be sent after a call has a call state of `eCallStatusReleased`.

Flow Control

I-Library responds to Flow Control XML messages from IVR Server in the following way:

When Load Sharing

- If a `Flow Control` XML message is received from IVR Server indicating that the flow control is `on`, I-Library will not send any `NewCall` or agent control messages to that IVR Server. I-Library will continue to send any XML messages that are part of an existing call to the IVR Server to which the `NewCall` message had already been sent. All other XML messages will be sent to other IVR Servers.
- If a `Flow Control` XML message is received from IVR Server indicating that the flow control is `off`, I-Library will start to send XML messages to that IVR Server. Any calls that are already in existence will continue to the

IVR Server to which their `NewCall` message had been sent. That IVR Server will be available for selection for any incoming `NewCall` and agent control messages.

When Not Load Sharing

- If a `Flow Control` XML message is received from IVR Server, indicating that the flow control is `on`, I-Library will be able to send only those XML messages that are part of an existing call that has already sent `NewCall` message to that IVR Server. All other API requests will receive a negative response.
- If a `Flow Control` XML message is received from IVR Server, indicating that the flow control is `off`, I-Library will start sending all API requests, as well as any agent control messages, to that IVR Server.

Genesys

Chapter

5 IVR API at a Glance

This chapter presents a condensed view of the IVR API and contains these sections:

The information for these functions is abbreviated. The functions themselves are presented in prototype form. For more information see the `interface.h` header file or the HTML API reference that is shipped with the IVR SDK C software.

Groups of IVR API Functions

IVR API functions can be grouped into functional categories.

Library Initialization and Reset Functions (page 62)

```
BOOL    ilInitiate(CPSTR);
BOOL    ilDeinitiate(void);
```

Opening and Closing Connections to IVR Server Functions (page 62)

```
BOOL    ilConnectionOpen(CPSTR, SOCKET_PORT, ULONG);
BOOL    ilConnectionOpenConfigServer(CPSTR, SOCKET_PORT, CPSTR,
                                     ULONG);
BOOL    ilConnectionOpenConfigServer80( CPSTR, SOCKET_PORT,
        SOCKET_PORT, CPSTR, SOCKET_PORT, CPSTR, ULONG;
BOOL    ilConnectionClose(void);
```

Network Socket Force for IVR Server Connection Functions (page 64)

```
ilRET   ilWatch(ULONG);
```

Get Reply to Previous Request Functions (page 65)

```
ilRET   ilGetReply(ilRQ, PSTR, int);
ilRET   ilGetReplyAny(ilRQ*, PSTR, int);
```

Notification Functions—Call Start/End (page 66)

```
ilRQ  ilSRqNoteCallStart(ilRQ, ilPORT, CPSTR, CPSTR, CPSTR, CPSTR);
ilRQ  ilSRqNoteCallEnd(ilRQ, ilPORT);
```

Telephony Functions—Requests (page 67)

```
ilRQ    ilSRqCallInit(ilRQ, ilPORT, CPSTR);
ilRQ    ilSRqCallComplete(ilRQ, ilPORT);
ilRQ    ilSRqCallConference(ilRQ, ilPORT, CPSTR);
ilRQ    ilSRqCallTransfer(ilRQ, ilPORT, CPSTR);
ilRQ    ilSRqCallConsultInit(ilRQ, ilPORT, CPSTR);
ilRQ    ilSRqCallConsultComplete(ilRQ, ilPORT);
ilRQ    ilSRqCallConsultConference(ilRQ, ilPORT);
ilRQ    ilSRqCallConsultTransfer(ilRQ, ilPORT);
ilRQ    ilSRqCallTransferKVList(ilRQ, ilPORT, CPSTR, CPSTR);
```

User Data–Processing Functions (page 70)

```
ilRQ    ilSRqUDataAddKD(ilRQ, ilPORT, CPSTR, CPSTR);
ilRQ    ilSRqUDataAddList(ilRQ, ilPORT, CPSTR);
ilRQ    ilSRqUDataGetKD(ilRQ, ilPORT, CPSTR);
ilRQ    ilSRqUDataDelKD(ilRQ, ilPORT, CPSTR);
ilRQ    ilSRqUDataDelAll(ilRQ, ilPORT);
ilRQ    ilSRqUDataGetAll( ilRQ, ilPORT);
```

CTI Object–Information Functions (page 73)

```
ilRQ    ilSRqGetCallInfo(ilRQ, ilPORT, ilCI_TYPE);
```

Call Data–Transfer Functions (page 74)

```
ilRQ    ilSRqCDT_Init(ilRQ, ilPORT, CPSTR, CPSTR, CPSTR, CPSTR);
ilRQ    ilSRqCDT_Cancel(ilRQ, ilPORT);
```

General–Purpose Functions (page 75)

```
ilRQ    ilSRqVersion(ilRQ, ilPORT, CPSTR);
BOOL    ilSetVersionNumber(CPSTR);
CPSTR   ilGetVersion(void);
int     ilGetCallStatus(ilPORT Port);
ilRET   ilGetProcessingState(void);
ilRQ    ilSRqToLog(ilRQ, ilPORT, CPSTR, CPSTR);
BOOL    ilSetLogHeader(CPSTR);
BOOL    ilSetTimeout(ULONG);
ULONG   ilGetTimeout(void);
```

Utility Functions (page 77)

```
long    ilGetLastError(ilERR_TYPE);
Long    ilGetLastPortError(iLPORT, ilERR_TYPE);
ilRET   ilLocalPrn(ilRQ, ilPORT, CPSTR,...);
ilRET   ilLocalPrnSelector(ilRQ, ilPORT, CPSTR,...);
CPSTR   ilPrnError(long);
CPSTR    ilGetParmValue( CPSTR, int );
```

Routing-Related Functions (page 80)

```
ilRQ    ilSRqRouteGet(ilRQ, ilPORT, CPSTR);
ilRQ    ilSRqRouteDone(ilRQ, ilPORT);
ilRQ    ilSRqRouteStart(ilRQ, ilPORT, CPSTR);
ilRQ    ilGetRequest(ilPORT, PSTR, int);
ilRET   ilSendReply(ilRQ, BOOL, CPSTR);
ilRQ    ilSRqRouteAbort(ilRQ, ilPORT);
```

Statistics-Related Functions (page 82)

```
ilRQ    ilSRqStatPeek(ilRQ, ilPORT, CPSTR);
ilRQ    ilSRqStatGet(ilRQ, ilPORT, CPSTR, CPSTR, CPSTR, CPSTR);
```

Outbound Calling Functions (page 82)

```
ilRQ    ilSRqGetDialOutData();
ilRQ    ilSRqDialOutDataInit(ilRQ, ilPORT, ilRQ);
ilRET   ilCallEstablished(ilPORT);
ilRET   ilDialOutError(ilPORT, ilDIALOUTERR);
ilRET   ilFailure(ilPORT, ilFAILURE);
```

IVR API Descriptions

This section briefly describes the purpose of each IVR API function, including descriptions of arguments. If a function's return values are obvious, there is no description.

Library Initialization and Reset Functions

Initiate

```
BOOL    ilInitiate(CPSTR);
```

The `ilInitiate()` function must be called before any other function in the library. As an argument, it takes the configuration name of the IVR driver. The `ilInitiate()` function returns the value `false` if there is a problem with the library and your driver should abort.

Deinitiate

```
BOOL    ilDeinitiate(void);
```

The `ilDeinitiate()` function is a noop, and exists only for backward compatibility. Genesys recommends that you not call it in your code.

Opening and Closing Connections to IVR Server Functions

ConnectionOpen

```
BOOL    ilConnectionOpen(CPSTR, SOCKET_PORT, ULONG);
```

The `ilConnectionOpen()` function opens a connection and logs in to IVR. Server in "`6.5 Mode.`" The 6.5 versions of the library operate in "`6.5 Mode.`" It then reads the configuration data that IVR Server sends in the response to the login request, and logs in to all IVR Servers that are configured as `load-sharing`. It also uses the configuration data to configure I-Library tables and processing options. As arguments, `ilConnectionOpen()` takes the name of the machine that hosts the IVR Server, the port number on which the IVR Server is listening, and the maximum number of milliseconds allowed to establish the connection. It returns the value `false` if I-Library cannot successfully log in to the IVR Server within the timeout value, or if the timeout value is negative. The function returns the value `true` if a connection to the IVR Server succeeds and handshakes are valid.

If the host parameter is `null`, or if the host is `unknown`, it will cause I-Library to exit.

The XML message generated for this API is `LoginReq`.

Note: The timeout value is saved and used as a timer whenever a connection to IVR Server is lost and must be re-opened. When attempting to re-connect to IVR Server, I-Library will use the `time_recon_is` value as a timer, to determine how often it should attempt to log into IVR Server. If the time specified by the timeout expires, I-Library will close and re-open the socket, and then continue to attempt to log in to IVR Server.

ConnectionOpenConfigServer

```
BOOL ilConnectionOpenConfigServer(CPSTR,SOCKET_PORT,CPSTR,ULONG);
```

The `ConnectionOpenConfigServer()` function operates in "`7.0 Mode,`" which incorporates new features such as dynamic configuration updates and centralized logging. "`7.0 Mode`" is not available on UnixWare platforms. It opens a connection to the Genesys Configuration Server in order to access the configuration data defined in the `Driver Application`, and it logs in to IVR Servers defined in the `load-sharing` option. It ignores the configuration data that IVR Server sends in the response to the login request. It uses the configuration data to configure I-Library tables and processing options. As arguments, `ilConnectionOpenConfigServer()` takes the name of the machine that hosts the Configuration Server, the port number on which the Configuration Server is listening, the name of the `Application` object that refers to the driver, and the maximum number of milliseconds allowed to establish the connection.

Note: The timeout value is saved and used as a timer whenever a connection to IVR Server is lost and must be re-opened. When attempting to re-connect to IVR Server, I-Library will use the `time_recon_is` value as a timer, to determine how often should attempt to log in to IVR Server. If the time specified by the timeout expires, I-Library will close and re-open the socket and then continue to attempt to log in to IVR Server.

The function exits if I-Library cannot successfully log into the Configuration Server. It returns the value `false` if I-Library cannot successfully log into the IVR Server within the timeout value or if the timeout value is negative. The function returns the value `true`, if a connection to the IVR Server succeeds and handshakes are valid.

If the host parameter is `null`, or if the host is `unknown`, I-Library exits.

The XML message generated for this API is `LoginReq`.

ConnectionOpenConfigServer80

```
Bool ilConnectionOpenConfigServer80( CPSTR, SOCKET_PORT,
SOCKET_PORT, CPSTR, SOCKET_PORT, CPSTR, ULONG );
```

The `ConnectionOpenConfigServer80` function operates just as the `ConnectionOpenConfigServer` function but with additional capabilities.

The first three parameters identify the Configuration Server to be connected to in order to start processing. The next two parameters need to be present if a backup or secondary Configuration Server is to be used. The last two parameters are the application name and the timeout value.

The first three parameters are the host, port, and client side port of the Configuration Server. You may leave the client side port as `0` (zero) if you do not want to specify a client side port.

If you want to indicate a backup Configuration Server, you may use the next two parameters to do so. As with the first two, they are the host and port of the backup Configuration Server.

It is only necessary to use this function if you want either a client side port defined for the Configuration Server or you want to provide a backup Configuration Server to use in the case that the primary Configuration Server is not available.

When I-Library does connect to a Configuration Server, it will ask that Configuration Server if it is configured.

ConnectionClose

```
BOOL    ilConnectionClose(void);
```

The `ilConnectionClose()` function is a noop, and exists only for backward compatibility. All connections to IVR Servers are closed when the I-Library exits. Genesys recommends that you not call `ilConnectionClose()` in your code.

Network Socket Force for IVR Server Connection Function

Watch

```
ilRET   ilWatch(ULONG);
```

The ilWatch() function ensures that I-Library has the opportunity to:

- Process replies to API messages
- Ensure that all sockets connected to the IVR Server are fully operational
- Ensure that the Configuration Server is fully operational
- Process keep-alive messages as necessary

The parameter is the minimum time in milliseconds that the application waits to expire before `ilWatch` returns. It may be `0` (zero), indicating a return as soon as all processing is completed.

The receipt of the `ilWatch` will not be logged unless the wait time is greater than 10 milliseconds.

The `ilWatch` is only necessary if call activity is low. During times when APIs are being sent to I-Library, the `ilWatch` function is initiated internally. If there is limited call activity, `ilWatch` should be used as often as the application wants:

- Socket connectivity to be checked.
- Keep-alive requests from IVR Server(s) to be responded to.

Get Reply to Previous Request Functions

GetReply

```
ilRET   ilGetReply(ilRQ, PSTR, int);
```

The `ilGetReply()` function retrieves reply data for a previous `ilSRq` function request. If the data is not available at the time of the call, it will wait up to the value specified in the last `ilSetTimeout`. If the `ilSetTimeout` value is `0` (zero), it will return immediately, without waiting for data. It will check for returned data during that time every 10 ms, until the `ilSetTimeout` value is reached. A number of bytes of the reply data string, as specified by `size,` are copied into buffer. If the size is smaller than the buffer size, the buffer is padded with zeroes; otherwise, it is not padded, and the buffer is not null terminated. As arguments, `ilGetReply()` takes a `ilRQ` value captured from the return of the previous `ilSRq*()` function, a pointer to a string buffer, and an integer that specifies the size of the string buffer, in bytes.

Return values include:

- `1` if the data is available in the buffer.
- `ilRET_BAD_ARGS` if the `ilrq` is `0`.
- `ilRET_TIMEOUT` if the `ilSetTimeout` value was exceeded before data became available.
- `ilRET_REQ_EXPIRED` if the `ilrq` is no longer an active request ID.

Note: A request ID goes inactive when a `ilGetReply` is issued for it and the data is available or when the call on which it was active ends.

For most APIs, the response buffer will contain the string `OK`. There are exceptions; however, these are described in those APIs that cause a different response string.

GetReplyAny

```
ilRET   ilGetReplyAny(ilRQ*, PSTR, int);
```

The `ilGetReplyAny()` function retrieves whatever is the current reply in the order stack, and writes the `ilRQ` number for that reply in the location specified by the first argument, which is a pointer to an `ilRQ` variable. Subsequent

arguments are pointers to a string buffer that will contain the text of the reply and the length of the buffer, in bytes.

Return values include:

- `1` if the data is available in the buffer.
- `ilRET_NO_REQUESTS` if there are no request IDs in the queue.

Notification Functions—Call Start/End

SRqNoteCallStart

```
ilRQ ilSRqNoteCallStart(ilRQ, ilPORT, CPSTR, CPSTR, CPSTR, CPSTR);
```

The `ilSRqNoteCallStart()` function notifies the IVR Server that a new call resides on a particular IVR resource that is specified by the `Port` argument. The IVR script should call this function before it calls any other telephony requests for the particular IVR channel (except for the `ilCallInit()` function, which performs the equivalent function). As arguments, `ilSRqNoteCallStart()` takes an `RqID` value (`ilRQ_ANY`), and an IVR port where the call resides. It can take four additional optional arguments: a string representation of a call ID that specifies a virtual call ID (PBX ID, switch ID), a string representation of a DNIS, a string representation of an ANI, and a string representation of a tag to be used for supporting the Call Data–Transfer (CDT) protocol. The I-Library checks both the call ID and the tag for CDT in order to ensure they are not null pointers, even though these two arguments are not used in the XML message that is sent to the IVR Server.

Return value include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log file.
- A positive request ID if successful.

`ilGetReply` will return a positive response immediately for this API. In order to determine whether the port is ready for further telephone activity, `ilGetCallStatus` can be used to determine when a `CallStatus ringing` or `CallStatus established` message has been received.

The XML message generated by this API is `NewCall`.

SRqNoteCallEnd

```
ilRQ    ilSRqNoteCallEnd(ilRQ, ilPORT);
```

The `ilSRqNoteCallEnd()` function notifies the IVR Server when call activity on a particular port has completed. In the IVR-behind-the-switch configuration, the IVR Server will request that the associated T-Server release the call. As arguments, `ilSRqNoteCallEnd()` takes an `ilRQ` value (`ilRQ_ANY`), and the IVR port where the call resides.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log file.
- A positive request ID if successful.

`ilGetReply` will return a positive response immediately for this API.

The XML message generated by this API is `EndCall`.

Telephony Functions—Requests

Note: The Telephony functions apply only to IVRs deployed in behind-the-switch mode.

SRqCallInit

```
ilRQ    ilSRqCallInit(ilRQ, ilPORT, CPSTR);
```

The `ilSRqCallInit()` function initiates a new call from the IVR port to a destination DN. As arguments, it takes an `ilRQ` value (`ilRQ_ANY`), the IVR port where the call resides, and a string representation of a DN (the directory number of the party to be dialed).

Note: This function will not work for the off-site switch option (IVR in-front-of-the-switch). If this function is used, the `ilSRqNoteCallStart` function should not be used for this same call.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log file.
- A positive request ID if successful.

`ilGetReply` will return a positive response after I-Library receives a `CallStatus Established` message.

The XML message generated by this API is `MakeCall`.

SRqCallComplete

```
ilRQ    ilSRqCallComplete(ilRQ, ilPORT);
```

The `ilSRqCallComplete()` function releases a call that is associated with the specified IVR port. It is equivalent to the `ilSRqNoteCallEnd()` function. As arguments, it takes an `ilRQ` value (`ilRQ_ANY`), and the IVR port where the call resides.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log file.
- A positive request ID if successful.

`ilGetReply` will return a positive response immediately for this API.

The XML message generated by this API is `EndCall`.

SRqCallConference

```
ilRQ    ilSRqCallConference(ilRQ, ilPORT, CPSTR);
```

The `ilSRqCallConference()` function makes a new conference call. As arguments, it takes an `ilRQ` value (`ilRQ_ANY`), the IVR port where the call resides, and a string representation of the destination DN for a new party.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log file.
- A positive request ID if successful.

`ilGetReply` will return a positive response after I-Library receives a `CallStatus ConfPartyAdd` message.

The XML message generated by this API is `OneStepConf`.

SRqCallTransfer

```
ilRQ    ilSRqCallTransfer(ilRQ, ilPORT, CPSTR);
```

The `ilSRqCallTransfer()` function completes a call transfer, causing the call on the specified port parameter to be released from both the original and consultation call. As arguments, it takes an `ilRQ` value (`ilRQ_ANY`), the IVR port where the call resides, and a string representation of the destination DN for a new party.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log file.
- A positive request ID if successful.

`ilGetReply` will return a positive response after I-Library receives a `CallStatus XferComplete` message.

The XML message generated by this API is `OneStepXfer`.

SRqCallTransferKVList

```
ilRQ    ilSRqCallTransferKVList(ilRQ, ilPORT, CPSTR, CPSTR);
```

The ilSRqCallTransferKVList() function does all that the ilSRqCallTransfer() function does and in addition enables the caller to also pass user data during the same function call. The last CPSTR variable is treated as a key-value pair list. The first character is considered to be the delimiter between values for the rest of the string. The usage rules are the same as for user data.

SRqCallConsultInit

```
ilRQ    ilSRqCallConsultInit(ilRQ, ilPORT, CPSTR);
```

The ilSRqCallConsultInit() function places an existing call on hold, and originates a consultation call. As arguments, it takes an ilRQ value (ilRQ_ANY), the IVR port where the call resides, and a string representation of the destination DN for a new party.

Return values include:

- ilRQ_ERR if there is no IVR Server available to process the call. ilRET_CONN_CLOSED is displayed in the log file.
- ilRQ_ERR if an exception occurs. ilRET_ERROR is displayed in the log file.
- A positive request ID if successful.

ilGetReply will return a positive response after I-Library receives a CallStatus Dialing message.

The XML message generated by this API is InitConf.

SRqCallConsultComplete

```
ilRQ    ilSRqCallConsultComplete(ilRQ, ilPORT);
```

The ilSRqCallConsultComplete() function releases a previously initiated consult call. As arguments, it takes an ilRQ value (ilRQ_ANY) and the IVR port where the call resides.

Return values include:

- ilRQ_ERR if there is no IVR Server available to process the call. ilRET_CONN_CLOSED is displayed in the log file.
- ilRQ_ERR if an exception occurs. ilRET_ERROR is displayed in the log file.
- A positive request ID if successful.

ilGetReply will return a positive response after I-Library receives a CallStatus Retrieved message.

The XML message generated by this API is RetrieveCall.

SRqCallConsultConference

```
ilRQ    ilSRqCallConsultConference(ilRQ, ilPORT);
```

The ilSRqCallConsultConference() function merges the original call and a consultation call into a conference call. As arguments, it takes an ilRQ value (ilRQ_ANY), and the IVR port where the consultation call resides.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log file.
- A positive request ID if successful.

`ilGetReply` will return a positive response after I-Library receives a `CallStatus Retrieved` message.

The XML message generated by this API is `CompleteConf`.

SRqCallConsultTransfer

```
ilRQ    ilSRqCallConsultTransfer(ilRQ, ilPORT);
```

The `ilSRqCallConsultTransfer()` function completes a call transfer. This causes the specified IVR port to be released from both the original and consultation calls. Parties participating in the original and consultation calls that reside on this port are merged into one call. As arguments, this function takes an `ilRQ` value (`ilRQ_ANY`), and the IVR port where the consultation call resides.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log file.
- A positive request ID if successful.

`ilGetReply` will return a positive response after I-Library receives a `CallStatus XferComplete` message.

The XML message generated by this API is `CompleteXfer`.

User Data-Processing Functions

SRqUDataAddKD

```
ilRQ    ilSRqUDataAddKD(ilRQ, ilPORT, CPSTR, CPSTR);
```

The `ilSRqUDataAddKD()` function attaches one user key-data combination to the active call that resides on the IVR port. There are certain restrictions on the data. The key cannot be a null string (“ ”). Any single character cannot be less than 0x20, except for \t, \r, and \n. There are also restrictions on the size of the key and data. User data strings may contain XML escape characters. Care is taken when translating these strings so that large strings do not cause performance impacts. See the *IVR Server System Administrator's Guide* for more information about this. As arguments, it takes an `ilRQ` value (`ilRQ_ANY`), the IVR port where the call resides, a string for the key, and a string for the data.

Return value include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if the key is null. `ilRET_BAD_ARGS` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log file.
- A positive request ID if successful.

`ilGetReply` will return a positive response after I-Library receives a `UDataResp` message.

The XML message generated by this API is `UDataSet`.

SRqUDataAddList

```
ilRQ    ilSRqUDataAddList(ilRQ, ilPORT, CPSTR);
```

The `ilSRqUDataAddList()` function adds a list of key-data pairs to the call that is residing on the specified IVR channel. The same restrictions that apply to the data for `SRqUDataAddKD` also apply to this API. The list must be structured with a delimiter between each key and value—for example,`:key:data:key2:data2:`. If the list contains a null character string in a position where a key is expected, that data will be skipped and the next string will be considered a key. As arguments, it takes an `ilRQ` value (`ilRQ_ANY`), the IVR port where the call resides, and a string that stores an ordered list.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if any key is null. `ilRET_BAD_ARGS` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log file.
- A positive request ID if successful

`ilGetReply` will return a positive response after I-Library receives a `UDataResp` message.

The XML message generated by this API is `UDataSet`.

SRqUDataGetKD

```
ilRQ    ilSRqUDataGetKD(ilRQ, ilPORT, CPSTR);
```

The `ilSRqUDataGetKD()` function requests the value of either the key that is specified, or a colon-separated list of key-value pairs that is specified by the colon-separated list of keys. As arguments, it takes an `ilRQ` value (`ilRQ_ANY`), the IVR port where the call resides, and a string that stores either the key, or a colon-separated list of keys.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.

- `ilRQ_ERR` if the key is null, or if it contains a semi-colon. `ilRET_BAD_ARGS` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log file.
- A positive request ID if successful.

`ilGetReply` will return a positive response after I-Library receives a `UDataResp` message. The get reply buffer will contain the data for the specified key or keys.

The XML message generated by this API is `UDataGet`.

SRqUDataGetAll

```
ilRQ    ilSRqUDataGetAll(ilRQ, ilPORT);
```

The `ilSRqUDataGetAll()` function requests all the currently known key-value pairs. As arguments, it takes an ilRQ value (`ilRQ_ANY`), and the IVR port where the call resides.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if the key is null, or if it contains a semi-colon. `ilRET_BAD_ARGS` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log file.
- A positive request ID if successful.

`ilGetReply` will return a positive response after I-Library receives a `UDataResp` message. The get reply buffer will contain a null delimited string of key-value pairs.

The XML message generated by this API is `UDataGetAll`.

SRqUDataDelKD

```
ilRQ    ilSRqUDataDelKD(ilRQ, ilPORT, CPSTR);
```

The `ilSRqUDataDelKD()` function requests the deletion of the key-data pair that is specified by the key. As arguments, it takes an `ilRQ` value (`ilRQ_ANY`), the IVR port where the call resides, and a string that stores the key.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log file.
- A positive request ID if successful.

`ilGetReply` will return a positive response after I-Library receives a `UDataResp` message.

The XML message generated by this API is `UDataDel`.

SRqUDataDelAll

```
ilRQ    ilSRqUDataDelAll(ilRQ, ilPORT);
```

The `ilSRqUDataDelAll()` function requests the deletion of all user data attached to the call that is associated with the specified IVR port. As arguments, it takes an `ilRQ` value (`ilRQ_ANY`), and the IVR port where the call resides.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log file.
- A positive request ID if successful.

`ilGetReply` will return a positive response after I-Library receives a `UDataResp` message.

The XML message generated by this API is `UDataDel`.

CTI Object–Information Functions

SRqGetCallInfo

```
ilRQ    ilSRqGetCallInfo(ilRQ, ilPORT, ilCI_TYPE);
```

The `ilSRqGetCallInfo()` function requests information about a type of call data, such as a connection ID or DN. If the call has been started, but at least a `ringing` event has not been received from IVR Server, this function will not send the request to the IVR Server, instead it return a `NULL` when `ilGetReply` is issued. If at least a `ringing` has been received from IVR Server, the request will be sent to IVR Server, and `ilGetReply` will wait for the response from IVR Server to fill in the response data. If the call ends and a new call has not been started on this port, and if `ilCI_TYPE` is set to `ilCI_LAST_EVENT_NAME`, it will set the response data for `ilGetReply` to whatever the last event received from IVR Server was. To enable an application to receive `Not Available` rather than `NULL` when no data is available from IVR Server, set the `compat65` configuration value must be set to `yes`. As arguments, it takes an `ilRQ` value (`ilRQ_ANY`), the IVR port where the call resides, and an `ilCI_TYPE`. See the `ilCI_TYPE` enums in the `interface.h` header file.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if the infoType is invalid. `ilRET_BAD_ARGS` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log file.
- A positive request ID if successful.

`ilGetReply` will return a positive response and the `GetReply` buffer will contain data as explained above.

The XML message generated by this API is `CallInfoReq`.

Call Data–Transfer Functions

SRqCDT_Init

```
ilRQ    ilSRqCDT_Init(ilRQ, ilPORT, CPSTR, CPSTR, CPSTR, CPSTR);
```

The `ilSRqCDT_Init()` function requests an access number or tag to make a Call Data Transfer (CDT) to a remote destination. Use the `ilGetReply()` function to retrieve access number information, an access number (indirect type), or tag value (depending on the type of CDT) from your driver's reply buffer. The CDT types are:

`Default`	Uses the configuration data already defined for multi-site routing.
`Indirect`	Enables the call to reach the destination DN by means of the Route Point. (I-Library translates this to `Route`.)
`DirectNT`	Used for direct dialing to the destination DN. (I-Library translates this to `DirectNotoken`.)
`DirectTO`	Used for direct dialing to the destination DN, with a tag going out from the client—the client generates the tag for CDT. (I-Library translates this to `Direct`.)
`DirectTI`	Used for direct dialing to the destination DN, with a tag coming in that is accessible by the client—CDT generates the tag. (I-Library translates this to `Direct`.)

If the IVR application passes a string that is not equal to one of these CDT types, the string, is sent as-is, to the IVR Server.

As arguments, this function takes an `ilRQ` value (`ilRQ_ANY`), the IVR port where the call resides, a string that specifies the directory number of the party to be dialed, a string that specifies the name of the remote T-Server that receives a call and attaches data to it, a string that specifies the type of Call Data Transfer request, and a string that specifies the tag used to mark a call. (The tag is currently not used, and it is not passed in the XML message.)

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if any input parameter is invalid. `ilRET_BAD_ARGS` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log file.
- A positive request ID if successful.

`ilGetReply` will return a positive response after I-Library receives an `AccessNumResp` message. The `GetReply` buffer will contain the access number.

The XML message generated by this API is `AccessNumGet`.

SRqCDT_Cancel

```
ilRQ    ilSRqCDT_Cancel(ilRQ, ilPORT);
```

The `ilSRqCDT_Cancel()` function cancels a previous `ilSRqCDT_Init()` function request. As arguments, it takes an `ilRQ` value (`ilRQ_ANY`), and the IVR port where the call resides.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log.
- A positive request ID if successful.

`ilGetReply` will return a positive response after I-Library receives an `AccessNumResp` message.

The XML message generated by this API is `AccessNumCancel`.

General-Purpose Functions

SRqVersion

```
ilRQ    ilSRqVersion(ilRQ, ilPORT, CPSTR);
```

The `ilSRqVersion()` function requests the version number of the I-Library or a named service. As arguments, it takes an `ilRQ` value (`ilRQ_ANY`), the IVR port where the call resides, and a string that specifies the service. Table 8 shows the service that is returned for a given `CPSTR` value.

Table 8: Service Version Returned with ilSRqVersion

CPSTR Value	Service Returned
Null or single space	Library version
Name of IVR	Driver version
All other values	I-Server version

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log.
- A positive request ID if successful.

`ilGetReply` will return a positive response immediately for this API. The `GetReply` buffer will contain the requested version.

SetVersionNumber

```
BOOL    ilSetVersionNumber(CPSTR);
```

The `ilSetVersionNumber()` function enables you to specify a version for your IVR driver. Its argument is a string representation of the version number for this driver.

Return values include:

- `false` if the version is `null`.
- `false` if an exception occurs. `ilRET_ERROR` is displayed in the log.

GetVersion

```
CPSTR   ilGetVersion(void);
```

The `ilGetVersion()` function enables you to retrieve the version of the I-Library. This will work even before issuing an `ilInitiate()`.

GetCallStatus

```
int     ilGetCallStatus(ilPORT);
```

The `ilGetCallStatus()` function returns the current call status on the port that is represented by the port parameter. The values that can be returned are defined as the `eCallStatus` enum in the `interface.h` file.

GetProcessingState

```
ilRET ilGetProcessingState(void);
```

The `ilGetProcessingState()` function returns the current operating state of the library. The values that can be returned are defined as the `I-LibraryState` enum in the `interface.h` file.

SRqToLog

```
ilRQ    ilSRqToLog(ilRQ, ilPORT, CPSTR, CPSTR);
```

The `ilSRqToLog()` function inserts an information string into the log file of `psService`. As arguments, it takes an `ilRQ` value (`ilRQ_ANY`), the IVR port where the call resides, a string that specifies the name of the service that has the log stream to be used, and a string that is the information to be placed into the log stream. Table 9 identifies where logged data is written based on the value of the first `CPSTR` parameter.

Table 9: Log File To Which Service Data Is Sent with SRqToLog

First CPSTR Value	Log File Written To
Name of IVR	IVR Server log file
All other values	Driver log file

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log.
- A positive request ID if successful.

`ilGetReply` will return a positive response immediately for this API.

The XML message generated by this API is `LogMsg` if the specified service is for IVR Server.

SetLogHeader

```
BOOL    ilSetLogHeader(CPSTR);
```

The `ilSetLogHeader()` function sets a header (product name, Copyright) for printing into the log file. This function is valid only in "`6.5 Mode.`" In "`7.x Mode,`" it is a noop. As arguments, it takes an `ilRQ` value (`ilRQ_ANY`); the IVR port where the call resides; and a string that can have any number of `\n` characters, and that stores the header information for the log file.

It returns `false` if an exception occurs. `ilRET_ERROR` is displayed in the log.

SetTimeout

```
BOOL    ilSetTimeout(ULONG);
```

The `ilSetTimeout()` function sets the timeout for the next calls of the `ilGetReply()` and `ilGetRequest()` functions. It takes an argument that specifies the timeout value, in milliseconds.

It returns `false` if the timeout value is negative.

GetTimeout

```
ULONG   ilGetTimeout(void);
```

The `ilGetTimeout()` function gets the current value, in milliseconds, of the timeout that was previously set by the `SetTimeout()` function.

Utility Functions

GetLastError

```
long    ilGetLastError(ilERR_TYPE);
```

The `ilGetLastError()` function returns the latest error code from the library. It is valid only before an API request from the following list is done. It also is not valid for checking the results of response messages from IVR Server; the `ilGetReply` API is used for that purpose.

The following APIs reset the last error:

- ilCallEstablished
- ilDialOutError
- ilFailure
- ilGetReply
- ilGetRequest
- ilInitiate
- ilSendReply
- ilSRqCallComplete
- ilSRqCallConference
- ilSRqCallConsultComplete
- ilSRqCallConsultConference
- ilSRqCallConsultInit
- ilSRqCallConsultTransfer
- ilSRqCallInit
- ilSRqCallTransfer
- ilSRqCDT_Cancel
- ilSRqCDT_Init
- ilSRqDialOutDataInit
- ilSRqGetCallInfo
- ilSRqGetDialOutData
- ilSRqNoteCallEnd
- ilSRqNoteCallStart
- ilSRqRouteAbort
- ilSRqRouteGet
- ilSRqRouteStart
- ilSRqStatGet
- ilSRqStatPeek
- ilSRqToLog
- ilSRqUDataAddKD
- ilSRqUDataAddlist
- ilSRqUDataDelAll
- ilSRqUDataDelKD
- ilSRqUDataGetKD
- ilSRqVersion

Pass in an ilERR_TYPE, either to specify an error number, or to print the error message to the log. If the input argument is ilET_NUMBER, the return value matches the error number for the error text. It will return ilRET_ERROR if an exception occurs.

GetLastPortError

```
long ilGetLastPortError(ilPORT port, ilERR_TYPE type);
```

The ilGetLastPortError() function returns the latest error code for the specified port. It also will print a string corresponding to the error in the log if type requests it. The error code represents the last error on the latest API request. It is valid only before an API request of the types listed in ilGetLastError API is done on that port. It also is not valid for checking the results of response messages from IVR Server; the ilGetReply API is used for that purpose. As arguments, it takes the IVR port where the call resides and an ilERR_TYPE that specifies whether or not to log an error in the log:

- If type = ilET_NUMBER, it will return the last error found.
- If type = ilET_TEXT, it will also log an error in the log that corresponds to the error found.

It returns ilRET_ERROR if an exception occurs.

LocalPrn

```
ilRET   ilLocalPrn(ilRQ, ilPORT, CPSTR,...);
```

The `ilLocalPrn()` function prints text directly to the local log. As arguments, it takes an `ilRQ` value (`ilRQ_ANY`), the IVR port where the call resides, and a string that is a format string that follows the rules for the `printf()` function, with the exception that a `\n` at the end of the string is ignored. Subsequent arguments match the format specifiers in the format string.

It will return `ilRET_ERROR` if an exception occurs.

LocalPrnSelector

```
ilRET   ilLocalPrnSelector(selector, ilRQi, lPORT, CPSTR,...);
```

The `ilLocalPrnSelector()` function prints directly to the local log based on a selection variable. Its arguments include `selector`, which is a string of zeroes and ones that indicate when the message should be printed. The string is compared to a string that is specified in the Configuration Layer and indicates which messages the user wants printed. If the `selector` string has a `1` in a position that matches a `1` in a position in the configured value in the Configuration Layer, the message is printed. As is the case with other arguments, it takes an `ilRQ` value (`ilRQ_ANY`); the IVR port where the call resides; and `psFmt`, a format string that follows the rules for the `printf()` function. Subsequent arguments match the format specifiers in the format string.

It will return `ilRET_ERROR` if an exception occurs.

PrnError

```
CPSTR   ilPrnError(long);
```

The `ilPrnError()` function returns a string pointer to the error message that is associated with an error code received from the `ilGetLastError()` function. Its argument is the error code.

It returns `ilRQ_ERR`, which is a null pointer, if an exception occurs. `ilRET_ERROR` is displayed in the log.

If the error number is unknown, it returns `Invalid error number` as the text.

GetParmValue

```
CPSTR ilGetParmValue( CPSTR, int );
```

The `ilGetParmValue()` function returns the value of the input key. If the value is a boolean, true or false, the second parameter can be set to `1` and the value will be translated to `true`, `1`, `yes`, or `on`, and `false` otherwise. If the key is not found, it returns `NULL`.

The key must be in a section in the `IVRDriver` application that is not a “known” section such as `ivr_server_interface`, `log`, `log_content`, or `security`.

Routing-Related Functions

SRqRouteGet

```
ilRQ    ilSRqRouteGet(ilRQ, ilPORT, CPSTR);
```

The `ilSRqRouteGet()` function retrieves the next service (route) from the Universal Routing Server (URS).

Note: Genesys recommends that you use the `RouteStart()` function instead of `RouteGet()` unless you have a specific need for this functionality (for instance, retrieving only the Routing Point).

As arguments, `ilSRqRouteGet()` takes an `ilRQ` value (`ilRQ_ANY`); the IVR port where the call resides; and a string that specifies the Routing Point at which the router has loaded a valid strategy, with only a next service (that is, a target) within that strategy.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log.
- A positive request ID if successful.

`ilGetReply` will return a positive response after I-Library receives a `RouteResp` message. The `GetReply` buffer will contain the route target.

The XML message generated by this API is `RouteRequest`.

SRqRouteDone

```
ilRQ    ilSRqRouteDone(ilRQ, ilPORT);
```

The `ilSRqRouteDone()` is a noop and exists only for backward compatibility. The IVR Server informs URS that the call has been routed (as requested in the `ilSRqRouteGet()` function). Genesys recommends that you not call this function in your code.

SRqRouteStart

```
ilRQ    ilSRqRouteStart(ilRQ, ilPORT, CPSTR);
```

The `ilSRqRouteStart()` function indicates to URS that a route sequence for `psRP` has started. The IVR Server will return treatments to the application via the `ilGetRequest()` function. As arguments, this function takes an `ilRQ` value (`ilRQ_ANY`); the IVR port where the call resides; and a string that specifies the Routing Point at which the router has loaded a valid strategy.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.

- ilRQ_ERR if an exception occurs. ilRET_ERROR is displayed in the log.
- A positive request ID if successful.

ilGetReply will return a positive response after I-Library receives a RouteResp message. The GetReply buffer will contain the target route.

The XML message generated by this API is RouteRequest.

GetRequest

```
ilRQ    ilGetRequest(ilPORT, PSTR, int);
```

The ilGetRequest() function checks for a request (treatment) from URS. It uses the value set in the last ilSetTimeout() to determine how long to wait if the treatment has not yet been received. If the ilSetTimeout value is 0 (zero), it will return immediately, without waiting for data. If no treatment is available within the timeout period, it returns a timeout error. As arguments, it takes the IVR port where the call resides, a string that stores the text of the treatment information, and the size of the buffer, in bytes.

Return values include:

- ilRQ_ERR if there is no IVR Server available to process the call. ilRET_CONN_CLOSED is displayed in the log file.
- ilRQ_ERR if the call on the port ends or a route response is received while it is waiting.
- ilRQ_ERR if an exception occurs. ilRET_ERROR is displayed in the log.
- ilTIMEOUT if no treatment is available within the timeout period.
- A positive request ID if successful.

The XML message generated by this API is TreatStatus.

SendReply

```
ilRET   ilSendReply(ilRQ, BOOL, CPSTR);
```

The ilSendReply() function sends a reply for a previously received treatment request (URS via IVR Server). As arguments, it takes an ilRQ value (ilRQ_ANY); the value true to print an OK message to the log, or the value false to print an error message to the log; and a pointer to the message to be sent as a reply (the contents depend on the treatment request).

Return values include:

- ilRET_CONN_CLOSED if there is no IVR Server available to process the call.
- ilRET_BAD_ARGS if the rqid is not active.
- ilRET_ERROR if an exception occurs.
- A positive request ID if successful.

The XML message generated by this API is TreatStatus.

SRqRouteAbort

```
ilRQ    ilSRqRouteAbort(ilRQ, ilPORT);
```

The `ilSRqRouteAbort()` function directs the I-Library to invalidate any future requests and replies for the Routing Point that is specified by the `ilSRqRouteStart()` function. Any events subsequently returned by the IVR Server are discarded.

Statistics-Related Functions

SRqStatPeek

```
ilRQ    ilSRqStatPeek(ilRQ, ilPORT, CPSTR);
```

The `ilSRqStatPeek()` function requests statistics from Stat Server. The `CurrNumberWaitingCalls` and `ExpectedWaitTime` statistics are supported. As arguments, this function takes an `ilRQ` value (`ilRQ_ANY`); the IVR port where the call resides; and a string that specifies the name of a statistic, as it is stored in the Configuration Layer.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log.
- A positive request ID if successful.

`ilGetReply` will return a positive response after I-Library receives a `StatResp` message. The `GetReply` buffer will contain the statistic value.

The XML message generated by this API is `PeekStatReq`.

SRqStatGet

```
ilRQ    ilSRqStatGet(ilRQ, ilPORT, CPSTR, CPSTR, CPSTR, CPSTR);
```

The `ilSRqStatGet()` function is provided for compatibility with previous versions of I-Library. The `ilSRqStatPeek()` function should be used instead. Genesys recommends that you not call this function in your code.

Outbound Dialing Functions

SRqGetDialOutData

```
ilRQ ilSRqGetDialOutData();
```

The `ilSRqGetDialOutData()` function requests the next number to be dialed by the IVR. There are no arguments.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call, `ilRET_CONN_CLOSED` is displayed in the log file.

- `ilRQ_ERR` if there are no numbers to dial and `ilRET_NO_REQUESTS` is displayed in the log.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log.
- A positive request ID if successful.

`ilGetReply` will return a positive response immediately. The `GetReply` buffer will contain:

- `DestNum`—The number to be dialed.
- `OrigNum`—The dialing number.
- `TimeToAnswer`—The amount of time to answer, in seconds.

SRqDialOutDataInit

```
ilSRqDialOutDataInit(ilRQ ilRq, ilPORT Port, ilRQ RqID_DialOut);
```

The `ilSRqDialOutDataInit()` function is equivalent to `NotifyCallStart`. It indicates that an outbound call has been made by the IVR, and it is in process. As arguments, it takes an `ilRQ` value (`ilRQ_ANY`), the IVR port where the call resides, and the request ID of the dial-out data.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log.
- `ilRQ_ERR` if the dial out data request ID is invalid. `ilRET_BAD_ARGS` is displayed in the log.
- A positive request ID if successful.

`ilGetReply` will return a positive response after I-Library receives a `CallStatus Dialing` message.

The XML message generated by this API is `DialOutInit`.

CallEstablished

```
ilCallEstablished(ilPORT ilPort);
```

The `ilCallEstablished()` function indicates that the outbound call made by the IVR has been established. As arguments, it takes the IVR port where the call resides.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log.
- `ilRET_OK` if successful.

The XML message generated by this API is `CallStatusEstablished`.

DialOutError

```
ilDialOutError (ilPORT ilPort, ilDIALOUTERR ilDialOutError);
```

The `ilDialOutError()` function indicates that a dialing error occurred when the IVR attempted to make the outbound call. As arguments, it takes the IVR port where the call resides and the type of error that occurred, `NotSupported`, `NoTrunks`, or `MiscError`.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log.
- `ilRET_OK` if successful.

The XML message generated by this API is `DialOutError`.

Failure

```
ilFailure(ilPORT ilPort, ilFAILURE ilFailure);
```

The `ilFailure()` function indicates that the outbound call made by the IVR has failed. As arguments, it takes the IVR port where the call resides and the type of failure that occurred, `Busy`, `NoAnswer`, or `ConnectFailed`.

Return values include:

- `ilRQ_ERR` if there is no IVR Server available to process the call. `ilRET_CONN_CLOSED` is displayed in the log file.
- `ilRQ_ERR` if an exception occurs. `ilRET_ERROR` is displayed in the log.
- `ilRET_OK` if successful.

The XML message generated by this API is `Failure`.

Appendix

7.0 Operating Mode

This appendix explains the 7.0 operating mode. It provides information about its configuration, initiation, and connections. This chapter contains the following sections:

Overview

I-Library has another operating mode that has not been available prior to release 7.2. The operating mode has traditionally been called "`7.0 Mode`." This mode has been available to the Genesys drivers that use the I-Library DLL, starting with release 7.0. In this mode, the I-Library has the ability to communicate with several of the Genesys Framework and Management Layer components such as Configuration Manager and Logging. This provides the I-Library with additional capabilities, such as dynamic configuration updates, centralized logging, and agent control to enable an orderly and controlled shutdown of the I-Library.

Notes:
- "7.0 Mode" is not available for the UnixWare version of the I-Library.
- Local Control Agent (LCA) must be installed, configured and running for the "7.0 Mode" to function.

Configuration

To enable the I-Library to be operated in "`7.0 Mode`," you must configure a `driver application` to provide the following additional information to the I-Library:

- System-level data that controls IVR Server communication.
- Logging data that controls how the Genesys logging libraries will log the I-Library-generated status information.

Some of this information is similar to what is configured in the IVR Object when the I-Library operates in "6.5 Mode." Additional information about how to configure this `driver application` is available in the *IVR Server System Administrator's Guide*.

Initiation

Initiation of the I-Library starts out as was explained in the `Start()` function in Chapter 2 on page 21, which must still call the `ilInitiate()` function (see Chapter 2 on page 21) and pass in the name of the IVR (as specified in the Configuration Layer) to initialize the I-Library for its use. The IVR Server still uses the IVR Object to maintain a list of the channels and corresponding DNs for every IVR, as described previously in Chapter 1 on page 13. In "`7.0 Mode,`" however, the configuration parameters (from the IVR object's `Annex` tab in the Configuration Layer) are no longer used by the I-Library. They may be present or not: this will have no effect on the operation of the I-Library.

Opening the IVR Server Connection

Previously, the `Start()` function called the `ilConnectionOpen()` function as described in Chapter 2 on page 21 to direct the I-Library to open a connection to the IVR Server whose host and port were identified in `ilConnectionOpen()`. In "`7.0 Mode,`" `ilConnectionOpenConfigServer()` (see Chapter 5 on page 59) is called by the `Start()` function. In this case, the host and port are those of the Genesys Configuration Manager. The I-Library will open a communication socket to Configuration Manager, and obtain all of the configuration data that it requires n order to operate successfully, as defined by the user through the driver application. In addition, the user can choose to operate the I-Library in a mode in which the I-Library controls agent activity. This provides the foundation for a controlled shutdown of the I-Library.

The format of the `ilConnectionOpenConfigServer()` function is:

```
BOOL ilConnectionOpenConfigServer(CPSTR host, SOCKET_PORT port,
CPSTR appName, ULONG ulTimeoutMS)
```

The host and port are those of Configuration Manager. The `appName` is the name of the `Driver Application`. The `ulTimeoutMS` is the same as was used in "`6.5 Mode.`"

A new API is available, `ilConnectionOpenConfigServer80`, which provides both client side port and backup Configuration Server capability.

In order to have the I-Library control agent activity, the `annex` tab of the IVR Object must include an `AgentControl` section that contains the option `LegacyMode` (see page 88) with a value of `false`.

Agent Control

When `LegacyMode` is set to `false`, agents will be controlled by the I-Library. During startup, the I-Library will obtain the list of agents configured in the Configuration Layer. The IVR object will be used to identify the ports that are to be evaluated as possible agents. If the `AutoLogin` section exists on the `Annex` tab on a port, and if at least `AgentId` and `Queue` information are provided, the port will be considered to have an agent configured, and the agent will be added to a list of agents that this I-Library will keep.

When a `MonitorInfo` XML message with `server sub-type` is received from IVR Server indicating that a switch is `up`, and when a `FlowControl` XML message is received from IVR Server, indicating that flow control is `off`, the agent activity thread will be started, in order to attempt to move all agents to their configured states. After all agents are in their configured states, the thread will end. If at least one agent does not reach its configured state, the thread will continue to attempt to move that agent. A configurable parameter, `DriverRetryTimeout` (see page 89) is used to determine how long to wait between attempts to move the agent.

Agent activity will be started for any of the following events:

- A `MonitorInfo`, `server sub-type` XML message is received from IVR Server. If the message indicates that the switch is down, all agent states will be set to `unknown`. If the message indicates that the switch is up, all agents will be moved to their configured states.
- An unsolicited `MonitorInfo, port sub-type` XML message is received from IVR Server. If the message indicates that the port is `disabled`, the agent configured in that port will be disabled.

If a configuration change is made for an agent, agent activity will attempt to move the agent to its newly configured state:

- If either the agent ID or queue is changed, the agent will be logged out and moved to its configured state.
- If the `SetLoggedIn` or `SetReady` is changed the agent will be logged out and moved to its new state.
- If `workmode` or `password` is changed, the information will be saved, but no activity will take place on the agent.

- If the port is disabled, the agent will be logged out and disabled. No further activity will take place for the agent until it is enabled.

When any of these events are dynamically received, the agent associated with the event will be added to the list of agents who need to be moved to a new state. Only the agents in this list will be processed. This will provide the best performance and minimize the impact to the telephony processing which is in progress. This is especially important to installations with a significant number of agents.

If a request to `shutdown is received` from Solution Control Interface (SCI), all agents will be logged out. The processing state will be changed as progress continues:

- `ilRET_ACTIVE` indicates that normal processing continues.
- `ilRET_SHUTDOWN_IN_PROCESS` indicates that a shutdown request has been received, and agents are being logged out.
- `ilRET_ALL_AGENTS_LOGGEDOUT` indicates that all agents are logged out, and calls are being monitored.
- `ilRET_NO_CALLS_IN_PROGRESS` indicates that all agents are logged out, and there are no calls in progress.

At this point this I-Library will continue to monitor calls and wait for the process to end.

IVR Annex Options

AgentControl Section

The options in the `AgentControl` section are used to specify which `AgentControl` values IVR Library expects, and what effect they have. Any values other than the ones described in this section are ignored.

LegacyMode

Default Value: `true`
Valid Values: `true, false`
Changes Take Effect: Immediately

Specifies whether IVR Server or IVR Driver controls agent activity for the IVR ports in the IVR object:

- If set to `true`, the IVR Sever controls the agent activity for the IVR ports in the IVR object, according to how those ports are configured.
- If set to `false`, the IVR Driver controls agent activity (`login/logout`, `Ready/NotReady` status, and so on.) for IVR ports. This enables graceful shutdown/startup.

DriverReadyWorkMode

Default Value: `ManualIn`
Valid Values: `ManualIn, AutoIn, Unknown`
Changes Take Effect: Immediately

Specifies the value that is used for `AttributeWorkMode` when IVR Server performs login operations. This value is sent in the `AgentReady` and `AgentNotReady` XML messages to IVR Server. This value is used exclusively in `TAgentReady`. If this option is set to `ManualIn`, when an agent state independently changes to `NotReady`, but is configured to be `Ready`, an `AgentReady` message will be sent to IVR Server.

DriverRetryTimeout

Default Value: `30`
Valid Values: Any integer
Changes Take Effect: Immediately

Specifies, in seconds, how long the driver waits before trying agent activity on a particular port, after receiving an error message from IVR Server for a previous agent control message on that port.

DriverIgnoreReady

Default Value: `false`
Valid Values: `true, false`
Changes Take Effect: Immediately

Determines whether IVR Driver attempts to use the `SetReady` parameter:

- If set to `true`, IVR Driver ignores the `SetReady` parameter.
- If set to `false`, IVR Driver attempts to set agents to the configured `SetReady` state.

Related Documentation Resources

The following resources provide additional information that is relevant to this software. Consult these additional resources as necessary.

IVR Server and IVR Drivers

- *IVR Interface Option 8.1 IVR Server System Administrator's Guide*, which describes/provides information about how to configure, install and operate IVR Server.
- *IVR Interface Option 8.1 IVR Drivers System Administrator's Guides*, which provide information about how to configure, install and operate the Genesys IVR Driver components.
- *IVR SDK 8.x C API* reference information, which is in HTML format (double-click index.html) in the documentation directory on the product DVD.
- The `interface.h` header file, which accompanies the IVR SDK product files in the `IVR_SDK` directory on the product DVD.
- The `IVRexample.c` file, on the Genesys documentation DVD, as a companion to this guide.
- Release Notes and Product Advisories for this product, which are available on the Genesys Technical Support website at `http://genesyslab.com/support`.

Genesys

- *Genesys Technical Publications Glossary,* which ships on the Genesys Documentation Library DVD and which provides a comprehensive list of the Genesys and computer-telephony integration (CTI) terminology and acronyms used in this document.

- *Genesys Migration Guide*, which ships on the Genesys Documentation Library DVD, and which provides documented migration strategies for Genesys product releases. Contact Genesys Technical Support for more information.

Information about supported hardware and third-party software is available on the Genesys Technical Support website in the following documents:

- *Genesys Supported Operating Environment Reference Manual*
- *Genesys Supported Media Interfaces Reference Manual*

Consult these additional resources as necessary:

- *Genesys Hardware Sizing Guide,* which provides information about Genesys hardware sizing guidelines for the Genesys 8.x releases.
- *Genesys Interoperability Guide,* which provides information on the compatibility of Genesys products with various Configuration Layer Environments; Interoperability of Reporting Templates and Solutions; and Gplus Adapters Interoperability.
- *Genesys Licensing Guide,* which introduces you to the concepts, terminology, and procedures relevant to the Genesys licensing system.
- *Genesys Database Sizing Estimator 8 Worksheets,* which provides a range of expected database sizes for various Genesys products.

For additional system-wide planning tools and information, see the release-specific listings of System Level Documents on the Genesys Technical Support website, accessible from the `system level documents by release` tab in the Knowledge Base `Browse Documents` Section.

Genesys product documentation is available on the:

- Genesys Technical Support website at `http://genesyslab.com/support`.
- Genesys Documentation wiki at `http://docs.genesyslab.com/`.
- Genesys Documentation Library DVD and/or the Developer Documentation CD, which you can order by e-mail from Genesys Order Management at `orderman@genesyslab.com`.

Document Conventions

This document uses certain stylistic and typographical conventions—introduced here—that serve as shorthands for particular kinds of information.

Document Version Number

A version number appears at the bottom of the inside front cover of this document. Version numbers change as new information is added to this document. Here is a sample version number:

80fr_ref_06-2008_v8.0.001.00

You will need this number when you are talking with Genesys Technical Support about this product.

Screen Captures Used in This Document

Screen captures from the product graphical user interface (GUI), as used in this document, may sometimes contain minor spelling, capitalization, or grammatical errors. The text accompanying and explaining the screen captures corrects such errors *except* when such a correction would prevent you from installing, configuring, or successfully using the product. For example, if the name of an option contains a usage error, the name would be presented exactly as it appears in the product GUI; the error would not be corrected in any accompanying text.

Type Styles

Table 10 describes and illustrates the type conventions that are used in this document.

Table 10: Type Styles

Type Style	Used For	Examples
Italic	• Document titles • Emphasis • Definitions of (or first references to) unfamiliar terms • Mathematical variables Also used to indicate placeholder text within code samples or commands, in the special case where angle brackets are a required part of the syntax (see the note about angle brackets on page 94).	Please consult the *Genesys Migration Guide* for more information. Do *not* use this value for this option. A *customary and usual* practice is one that is widely accepted and used within a particular industry or profession. The formula, $x + 1 = 7$ where x stands for . . .
Monospace font (Looks like `teletype` or `typewriter text`)	All programming identifiers and GUI elements. This convention includes: • The *names* of directories, files, folders, configuration objects, paths, scripts, dialog boxes, options, fields, text and list boxes, operational modes, all buttons (including radio buttons), check boxes, commands, tabs, CTI events, and error messages. • The values of options. • Logical arguments and command syntax. • Code samples. Also used for any text that users must manually enter during a configuration or installation procedure, or on a command line.	Select the `Show variables on screen` check box. In the `Operand` text box, enter your formula. Click `OK` to exit the `Properties` dialog box. T-Server distributes the error messages in `EventError` events. If you select `true` for the `inbound-bsns-calls` option, all established inbound calls on a local agent are considered business calls. Enter `exit` on the command line.
Square brackets ([ ])	A particular parameter or value that is optional within a logical argument, a command, or some programming syntax. That is, the presence of the parameter or value is not required to resolve the argument, command, or block of code. The user decides whether to include this optional information.	`smcp_server -host [/flags]`
Angle brackets (< >)	A placeholder for a value that the user must specify. This might be a DN or a port number specific to your enterprise. **Note:** In some cases, angle brackets are required characters in code syntax (for example, in XML schemas). In these cases, italic text is used for placeholder values.	`smcp_server -host <confighost>`

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

O

P

R

S

T

U

V