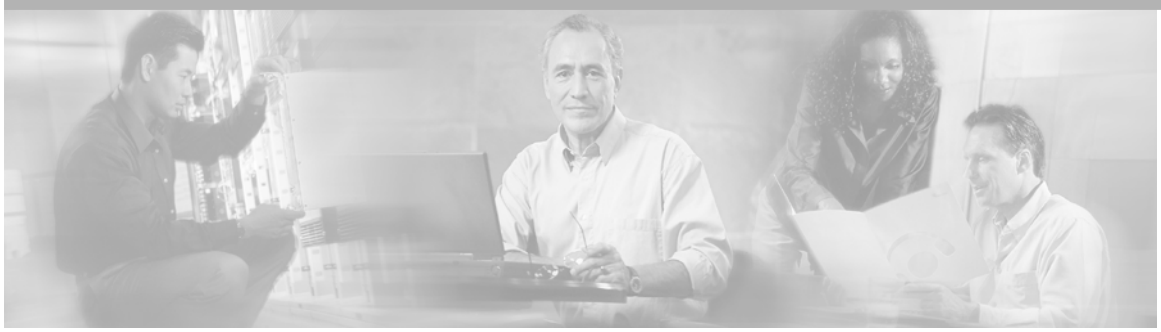
Cisco Systems



Tcl IVR 2.0 Programming Guide

Cisco IOS Release 12.3(14)T

Doc Release Date 3/28/2005

Corporate Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 526-4100

Customer Order Number:
Text Part Number: OL

THE SPECIFICATIONS AND INFORMATION REGARDING THE PRODUCTS IN THIS MANUAL ARE SUBJECT TO CHANGE WITHOUT NOTICE. ALL STATEMENTS, INFORMATION, AND RECOMMENDATIONS IN THIS MANUAL ARE BELIEVED TO BE ACCURATE BUT ARE PRESENTED WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. USERS MUST TAKE FULL RESPONSIBILITY FOR THEIR APPLICATION OF ANY PRODUCTS.

THE SOFTWARE LICENSE AND LIMITED WARRANTY FOR THE ACCOMPANYING PRODUCT ARE SET FORTH IN THE INFORMATION PACKET THAT SHIPPED WITH THE PRODUCT AND ARE INCORPORATED HEREIN BY THIS REFERENCE. IF YOU ARE UNABLE TO LOCATE THE SOFTWARE LICENSE OR LIMITED WARRANTY, CONTACT YOUR CISCO REPRESENTATIVE FOR A COPY.

The Cisco implementation of TCP header compression is an adaptation of a program developed by the University of California, Berkeley (UCB) as part of UCB's public domain version of the UNIX operating system. All rights reserved. Copyright © 1981, Regents of the University of California.

NOTWITHSTANDING ANY OTHER WARRANTY HEREIN, ALL DOCUMENT FILES AND SOFTWARE OF THESE SUPPLIERS ARE PROVIDED "AS IS" WITH ALL FAULTS. CISCO AND THE ABOVE-NAMED SUPPLIERS DISCLAIM ALL WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THOSE OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OR ARISING FROM A COURSE OF DEALING, USAGE, OR TRADE PRACTICE.

IN NO EVENT SHALL CISCO OR ITS SUPPLIERS BE LIABLE FOR ANY INDIRECT, SPECIAL, CONSEQUENTIAL, OR INCIDENTAL DAMAGES, INCLUDING, WITHOUT LIMITATION, LOST PROFITS OR LOSS OR DAMAGE TO DATA ARISING OUT OF THE USE OR INABILITY TO USE THIS MANUAL, EVEN IF CISCO OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

CCIP, the Cisco *Powered* Network mark, the Cisco Systems Verified logo, Cisco Unity, Follow Me Browsing, FormShare, Internet Quotient, iQ Breakthrough, iQ Expertise, iQ FastTrack, the iQ Logo, iQ Net Readiness Scorecard, Networking Academy, ScriptShare, SMARTnet, TransPath, and Voice LAN are trademarks of Cisco Systems, Inc.; Changing the Way We Work, Live, Play, and Learn, Discover All That's Possible, The Fastest Way to Increase Your Internet Quotient, and iQuick Study are service marks of Cisco Systems, Inc.; and Aironet, ASIST, BPX, Catalyst, CCDA, CCDP, CCIE, CCNA, CCNP, Cisco, the Cisco Certified Internetwork Expert logo, Cisco IOS, the Cisco IOS logo, Cisco Press, Cisco Systems, Cisco Systems Capital, the Cisco Systems logo, Empowering the Internet Generation, Enterprise/Solver, EtherChannel, EtherSwitch, Fast Step, GigaStack, IOS, IP/TV, LightStream, MGX, MICA, the Networkers logo, Network Registrar, *Packet*, PIX, Post-Routing, Pre-Routing, RateMUX, Registrar, SlideCast, StrataView Plus, Stratm, SwitchProbe, TeleRouter, and VCO are registered trademarks of Cisco Systems, Inc. and/or its affiliates in the U.S. and certain other countries.

All other trademarks mentioned in this document or Web site are the property of their respective owners. The use of the word partner does not imply a partnership relationship between Cisco and any other company.

Tcl IVR 2.0 Programming Guide
Copyright © 2004, Cisco Systems, Inc.
All rights reserved.

A-1

Preface

November 17, 2006

This guide describes Version 2.0 of the Tool Command Language (Tcl) Interactive Voice Response (IVR) Application Programming Interface (API). The Tcl IVR API can be used to create Tcl scripts that control calls coming in to or going out of a Cisco gateway. This guide provides an annotated example of a Tcl IVR script and instructions for testing and loading a Tcl IVR script.

Revision History

Release	Modification
12.3(14)T	• Added the following commands to Tcl IVR 2.0 for the Takeback and Transfer feature: – **consumeDigit** parameter to the **leg collectdigits** command. – **leg senddigit** command. – **leg sendhookflash** command. • Added the following to Tcl IVR 2.0 for HTTP support: – New and modified commands: **command export**, **media play** (modified), **media record** (modified), **modulespace**, **param read**, **param register**. – New information tags: ev_params, ev_module_handle, ev_module_subevent, ev_module_context. – New event: ev_synthesizer – Standard Tcl 8.3.4 commands supported are: cd, close, eof, fconfigure, file, fileevent, flush, gets, glob, open, package, pwd, read, seek. – HTTP commands supported are: config, geturl, formatQuery, reset, status, size, code, ncode, data, error, cleanup.
12.4(4)XC	Added the following commands to Tcl IVR 2.1 for the Extension Assigner feature: – **phone assign** command. – **phone query** command. – **phone unassign** command.

Command History

This section provides tables of changes and applicable Cisco IOS releases.

Table 1 Feature History: Commands

Cisco IOS Release	Command
12.2(11)T	**leg vxmldialog**
12.2(11)T	**leg vxmlsend**
12.2(11)T	**command terminate**
12.2(11)T	**aaa authentication**
12.2(11)T	**aaa authorization**

Table 1 Feature History: Commands (continued)

Cisco IOS Release	Command
12.2(11)T	**aaa accounting**
12.2(11)T	**clock**
12.2(11)T	**media play**
12.2(11)YT	**leg callerid**
12.2(11)YT	**leg consult abandon**
12.2(11)YT	**leg consult response**
12.2(11)YT	**leg consult request**
12.2(11)YT	**leg tranferdone**
12.2(15)T	**leg alert**
12.2(15)T	**leg disconnect_progind**
12.2(15)T	**leg setup_continue**
12.2(15)T	**leg progress**
12.2(15)T	**object create**
12.2(15)T	**object destroy**
12.2(15)T	**object append**
12.2(15)T	**object delete**
12.2(15)T	**object replace**
12.2(15)T	**object get**
12.2(15)T	**leg facility**
12.2(15)T	**log**
12.2(15)T	**media record**
12.3(4)T	**call close**
12.3(4)T	**handoff**
12.3(4)T	**handoff return**
12.3(4)T	**leg setup**
12.3(4)T	**sendmsg**
12.3(4)T	**service**
12.3(4)T	**set callinfo**
12.3(4)T	**subscription open**
12.3(4)T	**subscription close**
12.3(4)T	**subscription notify_ack**
12.3(4)T	**leg disconnect**
12.3(4)T	**call register**
12.3(4)T	**call unregister**
12.3(4)T	**call lookup**
12.3(4)T	**leg callerid**

Table 1 Feature History: Commands (continued)

Cisco IOS Release	Command
12.3(4)T	**leg collectdigits**
12.3(4)T	**aaa accounting set status**
12.3(4)T	**aaa accounting get status**
12.3(4)T	**aaa accounting probe**
12.3(4)T	**timer left**
12.3(4)T	**timer start**
12.3(4)T	**timer stop**
12.3(14)T	**leg senddigit**
12.3(14)T	**leg sendhookflash**
12.3(14)T	**command export**
12.3(14)T	**modulespace**
12.3(14)T	**param read**
12.3(14)T	**param register**

Table 2 Feature History: callInfo Parameters

Cisco IOS Release	callInfo Parameters
12.2(11)T	**guid**
12.2(11)T	**incomingGuid**
12.2(11)YT	**destinationNum**
12.2(11)YT	**originationNum**
12.2(11)YT	**accountNum**
12.2(11)YT	**redirectNum**
12.2(11)YT	**mode**
12.2(11)YT	**reroutemode**
12.2(11)YT	**transferConsultID**
12.2(11)YT	**notifyEvents**
12.2(11)YT	**originalDest**
12.2(15)T	**retryCount**
12.2(15)T	**interceptEvents**
12.2(15)T	**notifyEvents**
12.2(15)T	**previousCauseCode**
12.3(14)T	**consumeDigit**

Table 3 Feature History: Information Tags

Cisco IOS Release	Information Tag
12.2(11)T	**leg_rgn_noa**
12.2(11)T	**leg_rgn_npi**
12.2(11)T	**leg_rgn_pi**
12.2(11)T	**leg_rgn_si**
12.2(11)T	**leg_rgn_num**
12.2(11)T	**leg_rni_ri**
12.2(11)T	**leg_rni_orr**
12.2(11)T	**leg_rni_rc**
12.2(11)T	**leg_rni_rr**
12.2(11)T	**leg_ocn_noa**
12.2(11)T	**leg_ocn_npi**
12.2(11)T	**leg_ocn_pi**
12.2(11)T	**leg_ocn_num**
12.2(11)T	**leg_chn_noa**
12.2(11)T	**leg_chn_npi**
12.2(11)T	**leg_chn_num**
12.2(11)T	**leg_rnn_noa**
12.2(11)T	**leg_rnn_inn**
12.2(11)T	**leg_rnn_npi**
12.2(11)T	**leg_rnn_num**
12.2(11)T	**leg_rnr**
12.2(11)T	**leg_cdi_nso**
12.2(11)T	**leg_cdi_rr**
12.2(11)T	**leg_gno_ni**
12.2(11)T	**leg_cnn_noa**
12.2(11)T	**leg_cnn_npi**
12.2(11)T	**leg_cnn_pi**
12.2(11)T	**leg_cnn_si**
12.2(11)T	**leg_cnn_num**
12.2(11)T	**leg_gea_type**
12.2(11)T	**leg_gea_noa**
12.2(11)T	**leg_gea_npi**
12.2(11)T	**leg_gea_cni**
12.2(11)T	**leg_gea_pi**
12.2(11)T	**leg_gea_si**

Table 3 Feature History: Information Tags (continued)

Cisco IOS Release	Information Tag
12.2(11)T	**leg_gea_num**
12.2(11)T	**leg_cpc**
12.2(11)T	**leg_oli**
12.2(11)T	**leg_cid_ton**
12.2(11)T	**leg_cid_cid**
12.2(11)T	**leg_tns_ton**
12.2(11)T	**leg_tns_nip**
12.2(11)T	**leg_tns_cc**
12.2(11)T	**leg_tns_ns**
12.2(11)T	**leg_pci_instr**
12.2(11)T	**leg_pci_tri**
12.2(11)T	**leg_pci_dat**
12.2(11)T	**leg_fdc_parm**
12.2(11)T	**leg_fdc_fname**
12.2(11)T	**leg_fdc_instr**
12.2(11)T	**leg_fdc_dat**
12.2(11)T	**ev_vxmlevent**
12.2(11)T	**ev_vxmlevent_params**
12.2(11)T	**ev_status**
12.2(11)T	**ev_iscommand_done**
12.2(11)T	**ev_legs**
12.2(11)T	**last_command_handle**
12.2(11)T	**leg_guid**
12.2(11)T	**leg_incoming_guid**
12.2(11)T	**aaa_new_guid**
12.2(11)YT	**evt_consult_info**
12.2(11)YT	**evt_feature_report**
12.2(11)YT	**evt_feature_type**
12.2(11)YT	**evt_redirect_info**
12.2(11)YT	**evt_transfer_info**
12.2(11)YT	**leg_display_info**
12.2(11)YT	**leg_dn_tag**
12.2(15)T	**evt_gtd**
12.2(15)T	**evt_endpoint_address**
12.2(15)T	**evt_service_control_count**
12.2(15)T	**evt_service_control**

Table 3 Feature History: Information Tags (continued)

Cisco IOS Release	Information Tag
12.2(15)T	**evt_address_resolve_reject_reason**
12.2(15)T	**evt_address_resolve_term_cause**
12.2(15)T	**evt_last_event_handle**
12.2(15)T	**evt_facility_id**
12.2(15)T	**evt_facility_report**
12.2(15)T	**evt_gtd**
12.2(15)T	**evt_progress_indication**
12.2(15)T	**evt_status**
12.2(15)T	**gtd_attr_exists**
12.3(100)	**leg_rni_ri (removed)**
12.3(100)	**leg_rni_orr (removed)**
12.3(100)	**leg_rni_rc (removed)**
12.3(100)	**leg_rni_rr (removed)**
12.3(100)	**leg_ocn_noa (removed)**
12.3(100)	**leg_ocn_npi (removed)**
12.3(100)	**leg_ocn_pi (removed)**
12.3(100)	**leg_ocn_num (removed)**
12.3(100)	**leg_chn_noa (removed)**
12.3(100)	**leg_chn_npi (removed)**
12.3(100)	**leg_chn_num (removed)**
12.3(100)	**leg_rnn_noa (removed)**
12.3(100)	**leg_rnn_inn (removed)**
12.3(100)	**leg_rnn_npi (removed)**
12.3(100)	**leg_rnn_num (removed)**
12.3(100)	**leg_rnr (removed)**
12.3(100)	**leg_cdi_nso (removed)**
12.3(100)	**leg_cdi_rr (removed)**
12.3(100)	**leg_gno_ni (removed)**
12.3(100)	**leg_cnn_noa (removed)**
12.3(100)	**leg_cnn_npi (removed)**
12.3(100)	**leg_cnn_pi (removed)**
12.3(100)	**leg_cnn_si (removed)**
12.3(100)	**leg_cnn_num (removed)**
12.3(100)	**leg_gea_type (removed)**
12.3(100)	**leg_gea_noa (removed)**
12.3(100)	**leg_gea_npi (removed)**

Table 3 Feature History: Information Tags (continued)

Cisco IOS Release	Information Tag
12.3(100)	**leg_gea_cni (removed)**
12.3(100)	**leg_gea_pi (removed)**
12.3(100)	**leg_gea_si (removed)**
12.3(100)	**leg_gea_num (removed)**
12.3(100)	**leg_cpc (removed)**
12.3(100)	**leg_oli (removed)**
12.3(100)	**leg_cid_ton (removed)**
12.3(100)	**leg_cid_cid (removed)**
12.3(100)	**leg_tns_ton (removed)**
12.3(100)	**leg_tns_nip (removed)**
12.3(100)	**leg_tns_cc (removed)**
12.3(100)	**leg_tns_ns (removed)**
12.3(100)	**leg_pci_instr (removed)**
12.3(100)	**leg_pci_tri (removed)**
12.3(100)	**leg_pci_dat (removed)**
12.3(100)	**leg_fdc_parm (removed)**
12.3(100)	**leg_fdc_fname (removed)**
12.3(100)	**leg_fdc_instr (removed)**
12.3(100)	**leg_fdc_dat (removed)**
12.3(4)T	**leg_proto_headers**
12.3(4)T	**evt_handoff proto_headers**
12.3(4)T	**evt_handoff dnis**
12.3(4)T	**evt_handoff ani**
12.3(4)T	**evt_handoff argstring**
12.3(4)T	**evt_proto_content**
12.3(4)T	**evt_proto_content_type**
12.3(4)T	**evt_proto_headers**
12.3(4)T	**evt_status**
12.3(4)T	**evt_subscription_id**
12.3(4)T	**mod_all_handles**
12.3(4)T	**mod_handle**
12.3(4)T	**mod_handle_service**
12.3(4)T	**evt_msg**
12.3(4)T	**evt_msg_source**
12.3(4)T	**subscription_context**
12.3(4)T	**subscription_info**

Table 3 Feature History: Information Tags (continued)

Cisco IOS Release	Information Tag
12.3(4)T	**subscription_server_ipaddress**
12.3(4)T	**evt_disc_rsi**
12.3(4)T	**evt_disc_iec**
12.3(4)T	**evt_feature_report**
12.3(4)T	**evt_feature_param**
12.3(4)T	**evt_feature_type**
12.3(4)T	**evt_last_iec**
12.3(4)T	**media_timer_factor**
12.3(4)T	**set iec**
12.3(4)T	**evt_dest_handle**
12.3(4)T	**evt_handoff_legs**
12.3(4)T	**leg_type**
12.3(4)T	**aaa_accounting_last_sent**
12.3(4)T	**evt_aaa_status_info**
12.3(4)T	**evt_timer_name**
12.3(4)T	**leg_remote_media_ip_address**
12.3(4)T	**leg_remote_signaling_ip_address**

Table 4 Feature History: Events

Cisco IOS Release	Events
12.2(11)T	**ev_vxmldialog_done**
12.2(11)T	**ev_vxmldialog_event**
12.2(11)T	**leg_suppress_outgoing_auto_acct**
12.2(11)YT	**ev_consult_request**
12.2(11)YT	**ev_consult_response**
12.2(11)YT	**ev_consultation_done**
12.2(11)YT	**ev_transfer_request**
12.2(11)YT	**ev_transfer_status**
12.2(15)T	**ev_facility**
12.2(15)T	**ev_disc_prog_ind**
12.2(15)T	**ev_address_resolved**
12.2(15)T	**ev_alert**
12.2(15)T	**ev_connected**
12.2(15)T	**ev_proceeding**
12.2(15)T	**ev_progress**
12.3(4)T	**ev_accounting_status_ind**

Table 4 Feature History: Events (continued)

Cisco IOS Release	Events
12.3(4)T	**ev_named_timer**
12.3(4)T	**ev_feature**

Table 5 Feature History: Status Codes

Cisco IOS Release	Status Codes
12.2(11)T	**ls_016**
12.2(11)T	**vd_xxx—VoiceXML Dialog Completion Status**
12.2(11)YT	**cd_001 to cd_010**
12.2(11)YT	**cr_000 to cr_004**
12.2(11)YT	**cs_000 to cs_005**
12.2(11)YT	**ft_001 to ft_006**
12.2(11)YT	**ls_026**
12.2(11)YT	**ls_031 to ls_033**
12.2(11)YT	**ls_040 to ls_042**
12.2(11)YT	**ls_050 to ls_059**
12.2(11)YT	**ts_000 to ts_009**
12.2(15)T	**fa_000, fa_003, fa_007, fa_009, fa_010, fa_050 to fa_052**
12.3(4)T	**su_xxx, no_xxx, us_xxx, ul_xxx (000–010, 099)**

Audience

This document is a reference guide for developers writing voice application software for Cisco voice interfaces, such as the Cisco AS5x00 series universal access servers. Voice application developers may include:

- Independent software vendors (ISVs)
- Corporate developers
- System integrators
- Original equipment manufacturers (OEMs)

This document presumes:

- Tcl programming knowledge and experience

Although examples of how to create and use Tcl IVR scripts are provided in this document, this document is not intended to be a tutorial on how to write Tcl scripts.

Structure of This Guide

This guide contains the following chapters and appendixes:

- Chapter 1, "Overview" provides an overview of Interactive Voice Response (IVR), the Tool Command Language (Tcl), and version 2.0 of the Tcl IVR Application Programming Interface (API).
- Chapter 2, "Using Tcl IVR Scripts" contains information on how to create and use Tcl IVR scripts.
- Chapter 3, "Tcl IVR API Command Reference" provides an alphabetical listing of the Tcl IVR API commands.
- Chapter 4, "Information Tags" discusses identifiers that can be used to retrieve information about call legs, events, the script itself, the current configuration, and values returned from RADIUS.
- Chapter 5, "Events and Status Codes," describes events received and status codes returned by Tcl IVR scripts.
- Appendix , "Sample Scripts," provides some sample Tcl scripts.
- Glossary, presents an alphabetical listing of common terms used throughout this document.

Related Documents

- *Configuring Interactive Voice Response for Cisco Access Platforms:*

  http://www.cisco.com/univercd/cc/td/doc/product/access/acs_serv/as5400/sw_conf/ios_121/pull_ivr.htm
- *Service Provider Features for Voice over IP*:

  http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120t/120t3/voip1203.htm
- *Voice over IP for the Cisco AS5300:*

  http://www.cisco.com/univercd/cc/td/doc/product/access/nubuvoip/voip5300/index.htm
- *Voice over IP for the Cisco AS5800*:

  http://www.cisco.com/univercd/cc/td/doc/product/access/nubuvoip/voip5800/index.htm
- *Voice over IP for the Cisco 2600/Cisco 3600 Series:*

  http://www.cisco.com/univercd/cc/td/doc/product/access/nubuvoip/voip3600/index.htm
- *Configuring H.323 VoIP Gateway for Cisco Access Platforms*:

  http://www.cisco.com/univercd/cc/td/doc/product/software/ios121/121cgcr/multi_c/mcprt1/mcdvoip.htm
- *Prepaid Distributed Calling Card via Packet Telephony*:

  http://www.cisco.com/univercd/cc/td/doc/product/access/acs_serv/as5400/sw_conf/ios_121/pull0134.htm
- *RADIUS Vendor-Specific Attributes Implementation Guide:*

  http://cco/univercd/cc/td/doc/product/access/acs_serv/vapp_dev/vsaig3.htm
- *Tcl IVR API Version 1.0 Programmer's Guide:*

  http://cco/univercd/cc/td/doc/product/access/acs_serv/vapp_dev/tclivrpg.htm

- *Interactive Voice Response Version 2.0 on Cisco VoIP Gateways*:

  http://cco/univercd/cc/td/doc/product/software/ios121/121newft/121t/121t3/dt_skyn.htm
- *Enhanced Multilanguage Guide:*

  http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122newft/122t/122t2/ftmultil.htm
- *Cisco IOS Security Configuration Guide*, Release 12.2:

  http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122cgcr/fsecur_c/index.htm
- *Cisco IOS Tcl and VoiceXML Application Guide*

  http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122newft/122t/122t11/ivrapp/index.htm
- *Cisco VoiceXML Programmer's Guide*

  http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/rel_docs/vxmlprg/index.htm
- *Introduction to writing Tcl scripts:Tcl and the TK Toolkit, by John Ousterhout (published by Addison Wesley Longman, Inc)*
- Developer Support Agreement:

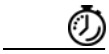
  http://www.cisco.com/warp/public/779/servpro/programs/ecosystem/devsup

Conventions

This guide uses the following conventions to convey instructions and information.

Convention	Description
boldface font	Commands and keywords.
italic font	Variables for which you supply values.
[ ]	Keywords or arguments that appear within square brackets are optional.
{x | y | z}	A choice of required keywords appears in braces separated by vertical bars. You must select one.
`screen font`	Examples of information displayed on the screen.
boldface screen font	Examples of information you must enter.
< >	Nonprinting characters, for example passwords, appear in angle brackets in contexts where italic font is not available.
[ ]	Default responses to system prompts appear in square brackets.

Note Means *reader take note*. Notes contain helpful suggestions or references to additional information and material.



Timesaver This symbol means *the described action saves time*. You can save time by performing the action described in the paragraph.

Caution This symbol means *reader be careful*. In this situation, you might do something that could result in equipment damage or loss of data.

Tip This symbol means *the following information will help you solve a problem*. The tips information might not be troubleshooting or even an action, but could be useful information, similar to a Timesaver.

Obtaining Documentation

The following sections provide sources for obtaining documentation from Cisco Systems.

World Wide Web

You can access the most current Cisco documentation on the World Wide Web at the following URL:

http://www.cisco.com

Translated documentation is available at the following URL:

http://www.cisco.com/public/countries_languages.shtml

Documentation CD-ROM

Cisco documentation and additional literature are available in a Cisco Documentation CD-ROM package, which is shipped with your product. The Documentation CD-ROM is updated monthly and may be more current than printed documentation. The CD-ROM package is available as a single unit or through an annual subscription.

Ordering Documentation

Cisco documentation is available in the following ways:

- Registered Cisco Direct Customers can order Cisco Product documentation from the Networking Products MarketPlace:

  http://www.cisco.com/cgi-bin/order/order_root.pl

- Registered Cisco.com users can order the Documentation CD-ROM through the online Subscription Store:

  http://www.cisco.com/go/subscription

- Nonregistered Cisco.com users can order documentation through a local account representative by calling Cisco corporate headquarters (California, USA) at 408 526-7208 or, in North America, by calling 800 553-NETS (6387).

Documentation Feedback

If you are reading Cisco product documentation on Cisco.com, you can submit technical comments electronically. Click the **Fax** or **Email** option under the "Leave Feedback" at the bottom of the Cisco Documentation home page.

You can e-mail your comments to bug-doc@cisco.com.

To submit your comments by mail, use the response card behind the front cover of your document, or write to the following address:

Cisco Systems
Attn: Document Resource Connection
170 West Tasman Drive
San Jose, CA 95134-9883

We appreciate your comments.

Obtaining Technical Assistance

Cisco provides Cisco.com as a starting point for all technical assistance. Customers and partners can obtain documentation, troubleshooting tips, and sample configurations from online tools by using the Cisco Technical Assistance Center (TAC) Web Site. Cisco.com registered users have complete access to the technical support resources on the Cisco TAC Web Site.

Cisco.com

Cisco.com is the foundation of a suite of interactive, networked services that provides immediate, open access to Cisco information, networking solutions, services, programs, and resources at any time, from anywhere in the world.

Cisco.com is a highly integrated Internet application and a powerful, easy-to-use tool that provides a broad range of features and services to help you to

- Streamline business processes and improve productivity
- Resolve technical issues with online support
- Download and test software packages
- Order Cisco learning materials and merchandise
- Register for online skill assessment, training, and certification programs

You can self-register on Cisco.com to obtain customized information and service. To access Cisco.com, go to the following URL:

http://www.cisco.com

Technical Assistance Center

The Cisco TAC is available to all customers who need technical assistance with a Cisco product, technology, or solution. Two types of support are available through the Cisco TAC: the Cisco TAC Web Site and the Cisco TAC Escalation Center.

Inquiries to Cisco TAC are categorized according to the urgency of the issue:

- Priority level 4 (P4)—You need information or assistance concerning Cisco product capabilities, product installation, or basic product configuration.
- Priority level 3 (P3)—Your network performance is degraded. Network functionality is noticeably impaired, but most business operations continue.
- Priority level 2 (P2)—Your production network is severely degraded, affecting significant aspects of business operations. No workaround is available.
- Priority level 1 (P1)—Your production network is down, and a critical impact to business operations will occur if service is not restored quickly. No workaround is available.

Which Cisco TAC resource you choose is based on the priority of the problem and the conditions of service contracts, when applicable.

Cisco TAC Web Site

The Cisco TAC Web Site allows you to resolve P3 and P4 issues yourself, saving both cost and time. The site provides around-the-clock access to online tools, knowledge bases, and software. To access the Cisco TAC Web Site, go to the following URL:

http://www.cisco.com/tac

All customers, partners, and resellers who have a valid Cisco services contract have complete access to the technical support resources on the Cisco TAC Web Site. The Cisco TAC Web Site requires a Cisco.com login ID and password. If you have a valid service contract but do not have a login ID or password, go to the following URL to register:

http://www.cisco.com/register/

If you cannot resolve your technical issues by using the Cisco TAC Web Site, and you are a Cisco.com registered user, you can open a case online by using the TAC Case Open tool at the following URL:

http://www.cisco.com/tac/caseopen

If you have Internet access, it is recommended that you open P3 and P4 cases through the Cisco TAC Web Site.

Cisco TAC Escalation Center

The Cisco TAC Escalation Center addresses issues that are classified as priority level 1 or priority level 2; these classifications are assigned when severe network degradation significantly impacts business operations. When you contact the TAC Escalation Center with a P1 or P2 problem, a Cisco TAC engineer will automatically open a case.

To obtain a directory of toll-free Cisco TAC telephone numbers for your country, go to the following URL:

http://www.cisco.com/warp/public/687/Directory/DirTAC.shtml

Before calling, please check with your network operations center to determine the level of Cisco support services to which your company is entitled; for example, SMARTnet, SMARTnet Onsite, or Network Supported Accounts (NSA). In addition, please have available your service agreement number and your product serial number.

Developer Support

Developers using this guide may be interested in joining the Cisco Developer Support Program. This program was created to provide you with a consistent level of support that you can depend on while leveraging Cisco interfaces in your development projects.

Tip A signed Developer Support Agreement is required to participate in this program. For more details go to http://www.cisco.com/go/developersupport or contact developer-support@cisco.com.

Overview

This chapter provides an overview of Interactive Voice Response (IVR), the Tool Command Language (Tcl), and version 2.0 of the Tcl IVR Application Programming Interface (API). This section includes the following topics:

IVR and Tcl

IVR is a term used to describe systems that collect user input in response to recorded messages over telephone lines. User input can take the form of spoken words or, more commonly, dual tone multifrequency (DTMF) signaling.

For example, when a user makes a call with a debit card, an IVR application is used to prompt the caller to enter a specific type of information, such as a PIN. After playing the voice prompt, the IVR application collects the predetermined number of touch tones (digit collection), forwards the collected digits to a server for storage and retrieval, and then places the call to the destination phone or system. Call records can be kept and a variety of accounting functions can be performed.

The IVR application (or script) is a voice application designed to handle calls on a *voice gateway*, which is a router equipped with voice features and capabilities.

The prompts used in an IVR script can be either static or dynamic:

- *Static prompts* are audio files referenced by a static URL. The name of the audio file and its location are specified in the Tcl script.
- *Dynamic prompts* are formed by the underlying system assembling smaller audio prompts and playing them out in sequence. The script uses an API command with a notation form (see the media play, page 3-48) to instruct the system what to play. The underlying system then assembles a sequence of URLs, based on the language selected and audio file locations configured, and plays them in sequence. This provides simple Text-to-Speech (TTS) operations.

  For example, dynamic prompts are used to inform the caller of how much time is left in their debit account, such as:

  "You have 15 minutes and 32 seconds of call time left in your account."

Note The above prompt is created using eight individual prompt files. They are: youhave.au, 15.au, minutes.au, and.au, 30.au, 2.au, seconds.au, and leftinyouraccount.au. These audio files are assembled dynamically by the underlying system and played as a prompt based on the selected language and prompt file locations.

The Cisco Interactive Voice Response (IVR) feature, available in Cisco IOS Release 12.0(6)T and later, provides IVR capabilities using Tcl 1.0 scripts. These scripts are signature locked, and can be modified only by Cisco. The IVR feature allows IVR scripts to be used during call processing. Cisco IOS software to perform various call-related functions. Starting with Cisco IOS Release 12.1(3), no longer is any Tcl script lock in place, so customers can create and change their own Tcl scripts.

Tcl is an interpreted scripting language. Because Tcl is an interpreted language, scripts written in Tcl do not have to be compiled before they are executed. Tcl provides a fundamental command set, which allows for standard functions such as flow control (if, then, else) and variable management. By design, this command set can be expanded by adding extensions to the language to perform specific operations.

Cisco created a set of extensions, called Tcl IVR commands, that allows users to create IVR scripts using Tcl. Unlike other Tcl scripts, which are invoked from a shell, Tcl IVR scripts are invoked when a call comes into the gateway.

The remainder of this document assumes that you are familiar with Tcl and how to create scripts using Tcl. If you are not, we recommend that you read *Tcl and the TK Toolkit,* by John Ousterhout (published by Addison Wesley Longman, Inc).

Tcl IVR API Version 2.0

This section describes the prerequisites, restrictions, benefits, features, and the developer support program for this application programming interface.

Prerequisites

In order to use the open Tcl IVR feature, you need the following:

- Cisco AS5300, Cisco AS5400, or Cisco AS5800
- Cisco IOS Release 12.1(3)T, or later

- Tcl Version 7.1 or later

Calls can come into a gateway using analog lines, ISDN lines, a VoIP link, or a Voice over Frame Relay (VoFR) link. Tcl IVR scripts can provide full functionality for calls received over analog or ISDN lines.

The functionality provided for calls received over VoIP or VoFR links varies depending on the release of Cisco IOS software being used. For example, if you are using Cisco IOS Release 12.0, you cannot play prompts or tones, and you cannot collect tones.

Note Tcl IVR API Version 2.0 is a separate product from Tcl IVR API Version 1.0.

Benefits

Tcl IVR API Version 2.0 has the following benefits:

- The scripts are event-driven and the flow of the call is controlled by a Finite State Machine (FSM), which is defined by the Tcl script.
- Prompts can be played over VoIP call legs.
- Digits can be collected over VoIP call legs.
- Real-Time Streaming Protocol (RTSP)-based prompts are supported (depending on the release of Cisco IOS software and the platform).
- Scripts can control more than two legs simultaneously.
- Call legs can be handed off between scripts.
- All verbs are nonblocking, meaning that they can execute without causing the script to wait, which allows the script to perform multiple tasks at once. See the following example code:

```
leg collect digits 1 callInfo
leg collect digits 2 callInfo
leg setup 295786 setupInfo $callID5
puts "\n This will be executed immediately i.e. before the collect digits or call
setup is actually complete"
```

  In the preceding script example, digit collection is initiated on legs 1 and 2 and a call setup process is started using the callID5 as the incoming leg. The script has issued each of the commands and will later receive events regarding their completion. None of these commands ever requires that any other command wait until it is finished processing.

Features Supported

Tcl IVR API Version 2.0 commands provide access to the following facilities and features:

- Call handling (setup, conferencing, disconnect, and so forth)
- Media playout and control (both memory-based and RTSP-based prompts)
- AAA authentication and authorization
- OSP settlements
- Call and leg timers
- Play tones
- Call handoff and return
- Digit collection

For more information, see Chapter 3, "Tcl IVR Commands."

Developer Support

Developers using this guide may be interested in joining the Cisco Developer Support Program. This new program has been developed to provide you with a consistent level of support that you can depend on while leveraging Cisco interfaces in your development projects.

A signed Developer Support Agreement is required to participate in this program. For more details, and access to this agreement, visit us at: http://www.cisco.com/warp/public/779/servpro/programs/ecosystem/devsup, or contact developer-support@cisco.com

Enhanced MultiLanguage Support

Beginning with Cisco IOS Release 12.2(2)T, a new feature has been introduced into Tcl IVR Version 2.0 that provides support for adding new languages and text-to-speech (TTS) notations to the core IVR infrastructure of the Cisco IOS gateway.

In the past, if you wanted an IVR application to do text-to-speech, you were limited to English, Spanish, and Chinese languages, and a fixed set of TTS notations. If an IVR application wanted to support more languages, it needed to do its own translation and include the language translation procedures with every Tcl IVR application that needed it.

With this new feature, you can make a new Tcl language module for any language and any set of TTS notations. You can test and deliver the module, and the audio files that go with it, as a language package, then document the language it delivers and the TTS notations it supports. When you configure this module on the gateway, any IVR application running on that gateway and using those TTS notations would work and speak that language.

For more information, refer to the *Enhanced Multi-Language Support for Cisco IOS Interactive Voice Response* document.

Note Tcl language modules are not Tcl IVR scripts. They are pure Tcl scripts that implement a specific Tcl language module interface (TLMI), so they must not use the Tcl IVR API extensions that are available for writing IVR scripts.

VoiceXML and IVR Applications

VoiceXML brings the advantages of web-based development and content to IVR applications. For more discussion on using VoiceXML with IVR applications, refer to the *Cisco IOS Tcl and VoiceXML Application Guide* and the *Cisco VoiceXML Programmer's Guide*.

Call Handoff in Tcl

Call handoff can best be understood when the concept of an application instance is first understood. In the Cisco IOS IVR infrastructure, an application instance is an entity that executes the application code and receives, creates, and manages one or more call legs to form a call, or to deliver a service to the user. The application instance owns and controls these call legs and receives all events associated with them. Although there can be exceptions, applications typically use a single application instance to deliver the services of a single call. Tcl IVR applications, when executing, act as one or more application instances.

Call Handoff is a term used to describe the act of transferring complete control of a call leg from one application instance to another. When handed off, all future events associated with that call leg will be received and handled by the target application instance.

Handoff can happen in several different ways, depending on whether the call leg needs to return to the source application instance of the handoff operation or not. A normal handoff application operation is similar to a *goto* event, with no automatic memory of a return address. The target cannot return the leg back to the source instance.

The *call app* operation is similar to a function call. The application instance performing the *call app* operation is saved on a stack and the target application instance can do a handoff return operation that returns the call leg to application instance on the top of the stack.

When doing a handoff of a call leg, any legs that are conferenced to that call leg are also handed off, even if they are not explicitly specified. When doing a handoff or a handoff return operation, an application instance can pass parameters as argument strings. Call handoff can take place between any combination of VoiceXML and Tcl IVR 2.0 applications.

The call handoff functionality allows a developer to write applications that may want to interact with each other for various purposes. This may be to use or leverage functionality in existing applications or to modularize a larger application into smaller application segments and use the handoff mechanism to coordinate and communicate between them. There may be times when the application developer need to leverage the functionality and features of both VoiceXML and Tcl IVR 2.0 in their applications. This may also be another application of the handoff operation.

Though handoff operations provide a certain amount of flexibility in achieving modularity and application interaction, they are limited when it comes to sharing control over a call leg. Only one application instance is in total control of the call leg and will receive events, which can prove to be limiting in certain scenarios. So, when considering a choice of mechanism for implementing applications involving both Tcl IVR 2.0 and VoiceXML, it is recommended that developers also consider *hybrid scripting* as an alternative.

Hybrid applications differ from call handoff operations. Hybrid applications are written using Tcl IVR scripts with VoiceXML dialogs either embedded or invoked in them. The Tcl IVR scripts are used for call control and the VoiceXML script is used for dialog management and they all run as part of one application instance allowing for a certain level of shared control of the call leg. Hybrid scripting is discussed in more detail in a later section.

Call Handoff in VoiceXML

The call handoff functionality in Cisco VoiceXML implementation is similar to the call handoff initiated by the *handoff appl* and *handoff callappl verbs* in Tcl IVR 2.0. For a discussion of call handoff in VoiceXML implementations, refer to the *Cisco VoiceXML Programmer Guide*.

Tcl/VoiceXML Hybrid Applications

Tcl IVR 2.0 and VoiceXML APIs each have their own strong points and some weak points. Tcl IVR 2.0 is very flexible when it comes to call control, able to describe multiple call legs, how they should be controlled, and how they should interwork. A weak point, however, is when it comes to user interface primitives being limited to **leg collectdigits** and **media play** commands.

VoiceXML on the other hand is both familiar and easy to use to design voice user interfaces, but is very limited in its call control capabilities. For example, VoiceXML dialog is good at IVR activities, such as collecting user input or playing prompts.

It would be advantageous, therefore, to use Tcl IVR 2.0 to describe the call legs , and the call flow and call control interactions between them, while using VoiceXML to describe user interface dialogs on one or more of the legs it is controlling.

Though it may be possible, to a limited extent, to use the handoff mechanism to have separate application instances in Tcl IVR 2.0 and use VoiceXML to deal with the call control and dialog aspects of the application, it is difficult to clearly partition, in time, the call control and dialog activities. This requires that the call control script and the dialog execution share control over the call leg, which is difficult to do in the handoff approach.

Cisco IOS Release 12.2(11)T introduces the ability for developers to use Tcl and VoiceXML scripts to develop hybrid applications. Tcl IVR 2.0 extensions allow Tcl applications to leverage support for ASR and TTS by invoking and managing VoiceXML dialogs from within Tcl IVR scripts. Hybrid applications can be developed using Tcl IVR for call control and VoiceXML for dialog management, allowing applications to use both Tcl IVR 2.0 and VoiceXML APIs, yet behave as a single application instance.

Hybrid scripting requires that some control sharing and precedence rules be established. In hybrid applications, the Tcl IVR 2.0 script is in control of the call and all of its call legs. It receives *ev_setup_indication* events for incoming call legs, and has the primitives to issue a *leg alert* or to accept the call leg through a **leg connect** command. It also has the primitives and event support to create outgoing call legs, bridging one or more call legs together, or other similar operations.

When the Tcl IVR script wants to communicate with the user on one of the call legs, it has two ways to do this. It can use the existing **leg collectdigits** and **media play** commands in native Tcl IVR 2.0 to play individual audio prompts and collect digits, or it can use the **leg vxmldialog** command to initiate the VoiceXML dialog operation on the leg. The **leg vxmldialog** command starts up a VoiceXML interpreter session on the call leg under the direct control of the Tcl IVR 2.0 script. The initial VoiceXML document that the session starts up could either be embedded in the Tcl IVR 2.0 script invoking it or it can simply refer to a VoiceXML document on a web server.

This VoiceXML session started on the leg is a normal VoiceXML session for the most part, but with the following exceptions:

- There are some synchronization primitives and mechanisms that have been added to allow information exchange between the VoiceXML dialog session and the Tcl IVR 2.0 call control script.
- VoiceXML supports some call control commands, such as the <*transfer*> and <*disconnect*> tags, which behave differently in this mode because the Tcl IVR 2.0 script should have complete control of all call control activities.

Communication Between VoiceXML and Tcl IVR 2.0 in Hybrid Applications.

When the Tcl IVR 2.0 script initiates a VoiceXML dialog on a call leg, it can pass an array of parameters to the **leg vxmldialog** command. These parameters becomes accessible from within the VoiceXML session through the *com.cisco.params.xxxxxx* variables. In the VoiceXML session, the *com.cisco.params* object gets populated with a information from the Tcl IVR array, where *xxxxx* is the index of the Tcl array.

When the VoiceXML dialog finishes, it can return some information back to the Tcl IVR script through the *namelist* attribute of the <*exit/*> tag. When the VoiceXML dialog finishes executing, the Tcl script receives the *ev_vxmldialog_done* event, which can carry with it the information returned in the exit tag. The event also carries with it a status code, which can be accessed through the *evt_status* information tag.

Apart from the start and end of a VoiceXML dialog, the Tcl script can send an intermediate message to a dialog in progress through the **leg vxmlsend** command. The event specified in the command is thrown inside VoiceXML interpreter and can be caught by a <*catch*> handler looking for that event. The command can also have a Tcl parameter array, whose information is accessible inside the VoiceXML catch handler through a scoped *_message.params.xxxxxx* variable, similar to *com.cisco.params.xxxx* described above.

Similarly, the VoiceXML interpreter environment or the executing document can send events to the Tcl script at various points. These events arrive at the Tcl script as *ev_vxmldialog_event* events. An executing VoiceXML document can use an <*object*> extension with *classid="builtin://com.cisco.ivrscript.sendevent"* to send an explicit message, with associated parameter information, to the parent Tcl script. If the VoiceXML document executes certain tags, such as <*disconnect*> or <*transfer*>*,* in the hybrid mode, that results in the Tcl script receiving an *ev_vxmldialog_event* event implicitly.

An *ev_vxmldialog_done* event or *ev_vxmldialog_event* event can come with two pieces of information:

- A VoiceXML-specific event name to differentiate the various reasons for the *ev_vxmldialog_done* or *ev_vxmldialog_event* event, which is accessible through the *evt_vxmlevent* information tag. This event name is a string in the form of *vxml.**. This indicates that the event name could be from the VoiceXML interpreter environment (vxml.session.*) or from the dialog executing in the VoiceXML interpreter (vxml.dialog.*). Examples of environment-level messages are *vxml.session.complete*, to indicate normal completion of a dialog, or *vxml.session.transfer,* to indicate that the document tried to execute a <*transfer*> tag, which is not supported in this mode of operation. If the document throws a *error.badfetch* message which is not caught and this causes the dialog to complete, or if the document uses the <*object*> send tag to send Tcl an explicit message, *evt_vxmlevent* will contain a *vxml.dailog.** string.
- A parameter array of information that is accessible through the *evt_vxmlevent_params* information tag.

Hybrid Mode and VoiceXML Call Control Tags

In the hybrid mode, the VoiceXML <*disconnect*> tag does not disconnect the call leg. Instead, a *vxml.session.disconnect* event is sent to the Tcl IVR script. From a VoiceXML execution perspective, a <disconnect> is emulated, throwing a disconnect event and then continuing execution. The dialog will not be able to play prompts or collect input from this point onwards.

When the user hangs up, a <disconnect> is again emulated, as above. But the leg is not disconnected yet. The Tcl script receives the *ev_disconnected* event as part of the control events, then has the responsibility of either terminating, or waiting for the termination of the dialog, and then disconnecting the leg.

When the document executes a <*transfer*> tag, this results in the following:

- A *vxml.session.transfer* event is sent by the VoiceXML environment to the Tcl script.
- The VoiceXML environment will throw an *error.unsupported.transfer* event at the VoiceXML session, which can be caught. If not caught, the default handler causes the termination of the dialog, resulting in an eventual *ev_vxmldialog_done* event to the Tcl script.

SendEvent Object

Recorded objects are represented as audio object variables in VoiceXML/JAVA scripting. In Tcl, which is totally text based, objects are represented as a *ram://XXXXX* URI. Tcl array elements that have a value of *ram://XXX* are available as audio variables or objects in VoiceXML. A similar reverse transformation happens when information is passed from VoiceXML to the Tcl script.

Tcl IVR Call Transfer Overview

Tcl IVR scripts can be used to provide blind- and consultation-transfer support for a variety of call transfer protocols. This section provides some background information about call-transfer terminology and usage scenarios as related to Tcl IVR applications. It also describes the call-transfer capabilities of each supported protocol and how these protocols can be interworked when the endpoints involved in the transfer use different signaling protocols.

Call Transfer Terminology

Transfer participants

A call transfer typically involves three participants:

- Transferor (XOR)—The endpoint that initiates the transfer.
- Transferee (XEE)—The endpoint that is transferred to different destination.
- Transfer target (XTO)—The endpoint that the transferee is transferred to.

Transfer Trigger

A call-transfer trigger is the mechanism a transferor endpoint uses to initiate a call-transfer procedure. This is normally a hookflash event for analog phones, or a button or softkey on an IP phone registered with the Cisco IOS voice gateway operating in Cisco CallTw55tieManager Express (CME) mode.

Transfer Commit

A transfer commit is the action a transferor endpoint takes when it wants to connect the transferee and transfer target endpoints, possibly after consulting with the transfer target endpoint. For analog phones and Cisco CME IP phones, the transfer commit is usually performed by hanging up the phone. When a Tcl IVR script receives a transfer-commit indication, it normally attempts to send a transfer request to the transferee call leg.

Supported Tcl IVR Call Transfer Script

Cisco provides an official Tcl IVR script that supports the H.450 call transfer scenarios discussed in the remainder of this section. This script is available in the Cisco CallManager Express (CME) zip files found at http://www.cisco.com/cgi-bin/tablebuild.pl/ip-key. The current version of the script is named *app_h450_transfer.2.0.0.3.tcl*. Refer to the README file associated with the script for more details.

Call Transfer Support in the Cisco IOS Default Session Application

Call transfer support has been added to the default voice session application in the 12.2(15)ZJ Cisco IOS release. The default application now provides H.450 and SIP transferee and transfer target functionality for blind and consultation transfers. It also provides H.450 and SIP blind and consultation transferor support for IP phones connected to the Cisco IOS gateway while operating in Cisco CallManager Express (CME) mode.

Note The enhanced default session application does not provide support for transfer initiation using an analog phone connected to the Cisco IOS gateway. This functionality is provided in the app_h450_transfer.2.0.0.3.tcl script mentioned above or can be implemented in a custom Tcl IVR application.

Custom Tcl IVR Call Transfer Scripts

The Cisco IOS default session application and app_h450_transfer.tcl script described above can be used to support many typical call transfer scenarios. In cases where a variant of this functionality is needed, a custom Tcl IVR script can be written. The call-transfer-sample.zip file on the Developer Support Central page contains sample Tcl IVR scripts and associated documentation that can be used as a guide in writing such a script.

Further assistance in Tcl IVR script writing can be obtained by joining the Cisco Developer Support program. This program provides a central resource for all development needs. Members of the program gain access to all available product and documentation downloads, bug reports, sample scripts, and frequently asked questions to facilitate development efforts.

The Developer Support engineers have subject matter expertise in Cisco interfaces and protocols. This team is dedicated to helping customers, and Cisco AVVID Partner Program and other ecosystem members, to use Cisco application programming interfaces (APIs) in their development projects. In addition to the benefits accessed from Cisco.com, the program provides an easy process to open, update, and track issues through Cisco.com. The Developer Support Agreement, which defines support commitments, fees, and available options, can be obtained from the Cisco Developer Support Web site at http://www.cisco.com/warp/public/570/.

Call Transfer Scenarios

There are many call transfer scenarios to consider when writing a Tcl IVR script. This subsection describes several such scenarios involving one, two, or three Cisco IOS voice gateways. To illustrate the call transfer scenarios, each description that follows includes the following diagrams:

- The first diagram shows the two-party call before the transfer.
- The second diagram shows a blind call transfer in progress.

- The third diagram shows a consultation transfer in progress.
- The fourth diagram shows the final call after a successful blind or consultation transfer.

Depending on the specific requirements, a script can be written to provide support for one or more of the scenarios that follow. In some cases, such as the consultation transfer scenario shown in Figure 1-7, two independent instances of the script may be active on the same gateway.

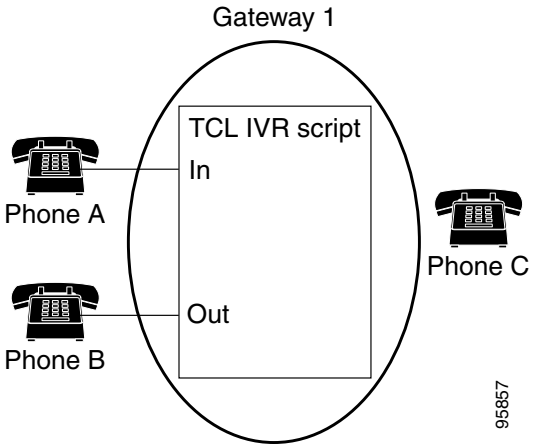
In the figures that follow, the labels XOR, XEE, and XTO designate the role each call leg plays in the call transfer. The IN and OUT labels track the incoming and outgoing call legs during a two-party call. This allows a script to keep track of the call leg topology and determine what action to take when an event is received.

In all scenarios described here, the original two-party call between phone A and phone B is already established. Phone A is the transferor endpoint (XOR), phone B is the transferee endpoint (XEE), and phone C is the transfer target endpoint (XTO). Transferor phone A is either an analog FXS phone or an IP phone registered with the Cisco IOS voice gateway operating in Cisco CallManager Express (CME) mode.

One Gateway Scenario with Analog Transferor

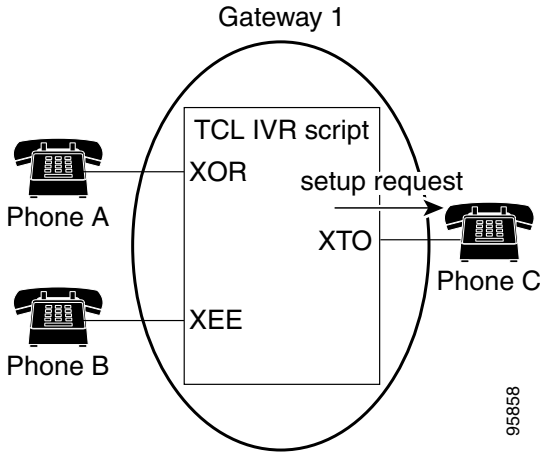
The first call transfer scenario is one in which phones A, B, and C are connected to the same gateway, as shown in Figure 1-1. In this case, all transferor, transferee, and transfer-target functionality is provided by a single instance of the Tcl IVR script.

Figure 1-1 Single GW: Analog XOR Before Transfer



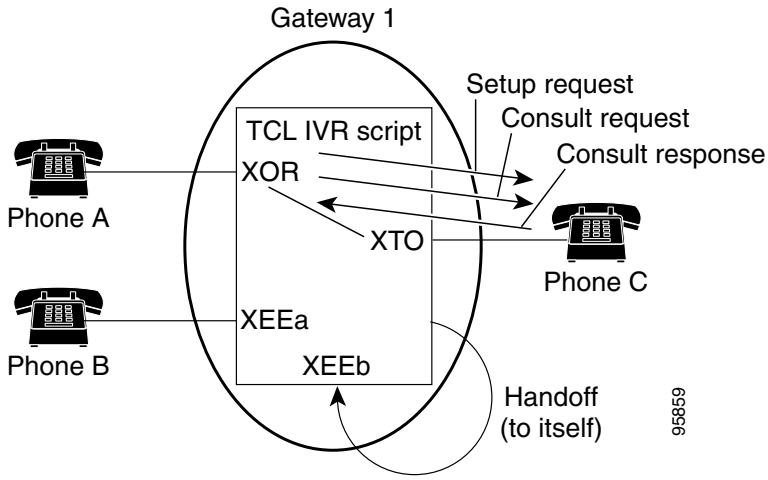
To initiate a blind transfer, the analog phone user presses hookflash, enters the transfer destination, and then hangs up. The script then places a regular call to the transfer target, connects the transferee and transfer-target call legs, then disconnects the transferor call leg. See Figure 1-2.

Figure 1-2 Single GW: Analog XOR Blind Transfer



To initiate a consultation transfer, the analog phone user presses hookflash and enters the transfer destination number. The script then places a call to the transfer target so that phone A can consult with phone C. When the user commits the transfer (by hanging up), the script requests a consultation ID from the transfer target (phone C). Because phone C is a local analog phone, the gateway generates a local consultation ID and registers it to this script instance. The script then places the outbound transfer call to phone C that includes this consultation ID. Because the consultation ID is registered to this script instance, the transferee call leg is handed off to this same script. See Figure 1-3.

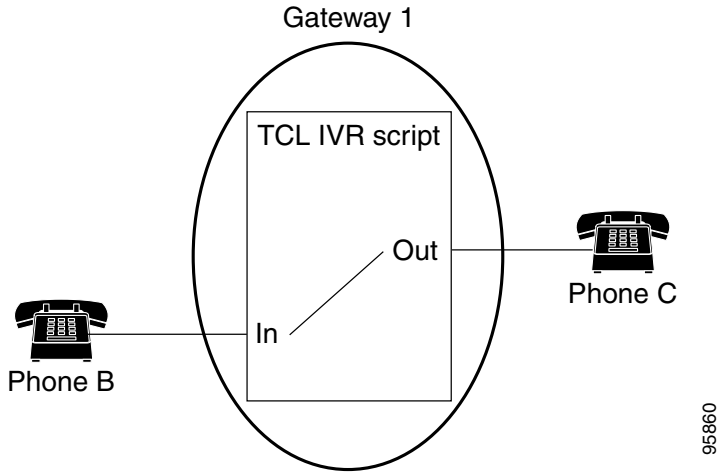
Figure 1-3 Single GW: Analog XOR Consultation Transfer



When the script receives the handoff event, it bridges the transferee and transfer-target legs and releases the transferor. See Figure 1-4.

Note In this single gateway scenario, it would be possible to simplify the call flow and avoid having the script hand off the transferee call leg to itself; however, using the handoff mechanism is the preferred approach as it also works in the multi-gateway scenarios described below.

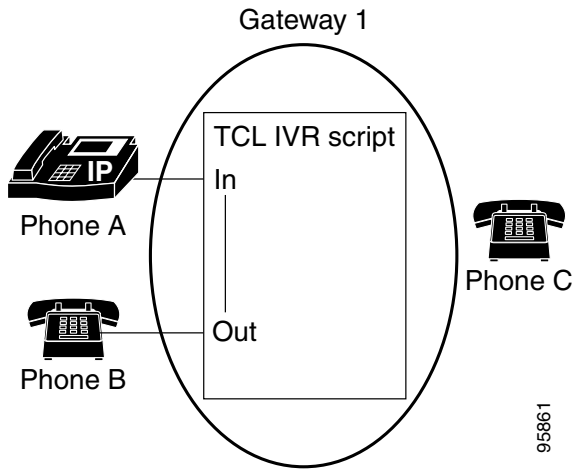
Figure 1-4 Single GW: Analog XOR After Transfer



One Gateway Scenario with Cisco CME IP Phone Transferor

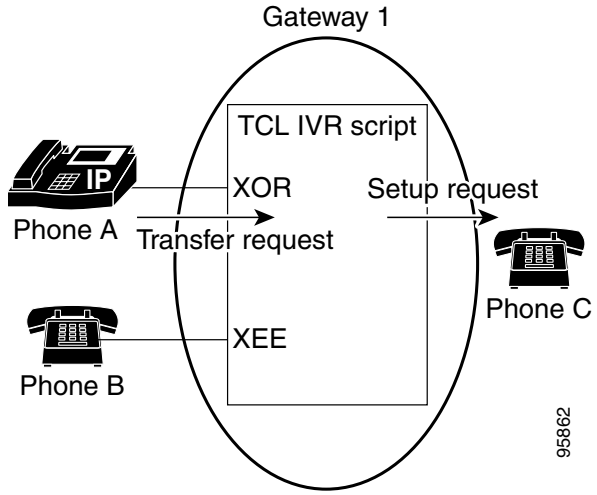
In this transfer scenario, phones A, B, and C are all connected to the same gateway. See Figure 1-5. In this case the transferor, transferee, and transfer-target functionality is provided by one or two instances of the Tcl IVR script.

Figure 1-5 Single GW: Cisco CME IP Phone XOR Before Transfer



To initiate a blind transfer, the IP phone user presses the transfer button on the phone and enters the transfer destination. The script receives a transfer request event from the phone, places a regular call to the transfer target, and connects the transferee and transfer target call legs. It then disconnects the transferor call leg. See Figure 1-6.

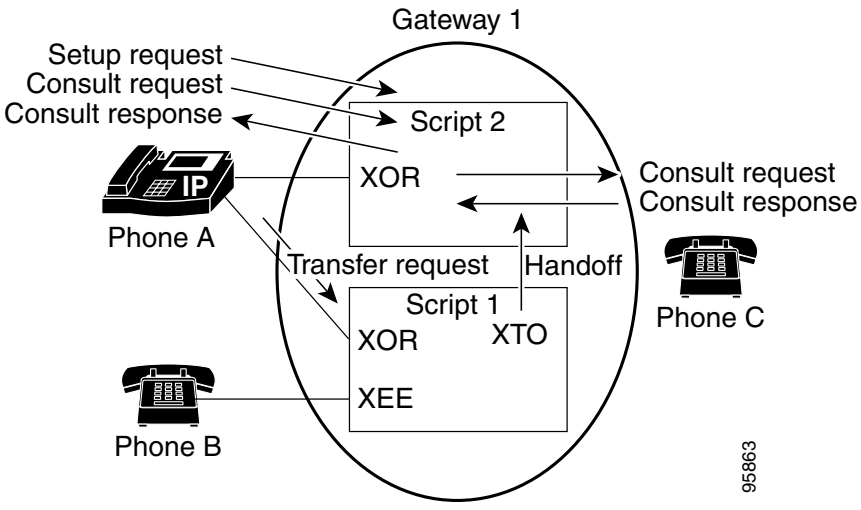
Figure 1-6 Single GW: Cisco CME IP Phone XOR Blind Transfer



To initiate a consultation transfer, the IP phone user presses the transfer button on the phone and enters the transfer-destination number. The IP phone uses a separate line to place a call to the transfer target. This call is independently handled by a second instance of the Tcl IVR script running on the gateway, which treats the call as a normal two-party call, unaware it is a consultation call.

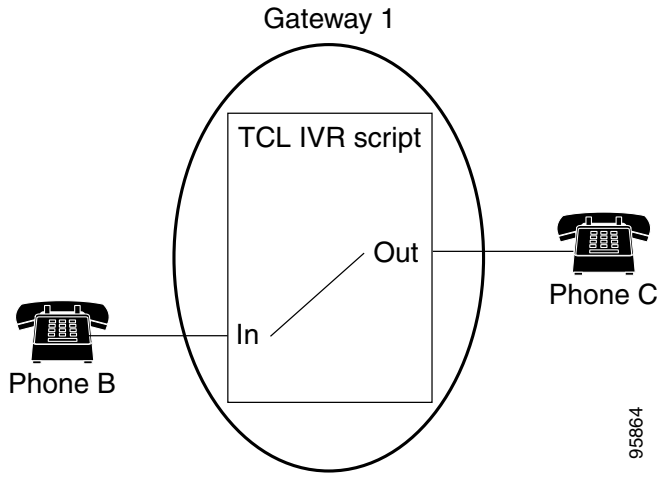
After consulting with the transfer target, the user commits the transfer by hanging up and the second script instance receives a consultation request from IP phone A. Because phone C is a local analog phone, the gateway generates a local consultation ID and registers it to this script instance. The script then sends a consultation response to IP phone A that includes this consultation ID. Next, the first script instance receives a transfer request from IP phone A that includes the consultation ID it received from the second script instance. See Figure 1-7.

Figure 1-7 Single GW: Cisco CME IP Phone XOR Consultation Transfer



This script instance then places the outbound transfer call to phone C that includes the consultation ID. Because the consultation ID is registered to the second script instance, the transferee call leg is handed off to the second script instance. The second script instance receives the handoff event and bridges the transferee and transfer-target legs. The first script instance releases the transferor call leg. See Figure 1-8.

Figure 1-8 Single GW: Cisco CME IP Phone XOR After Transfer



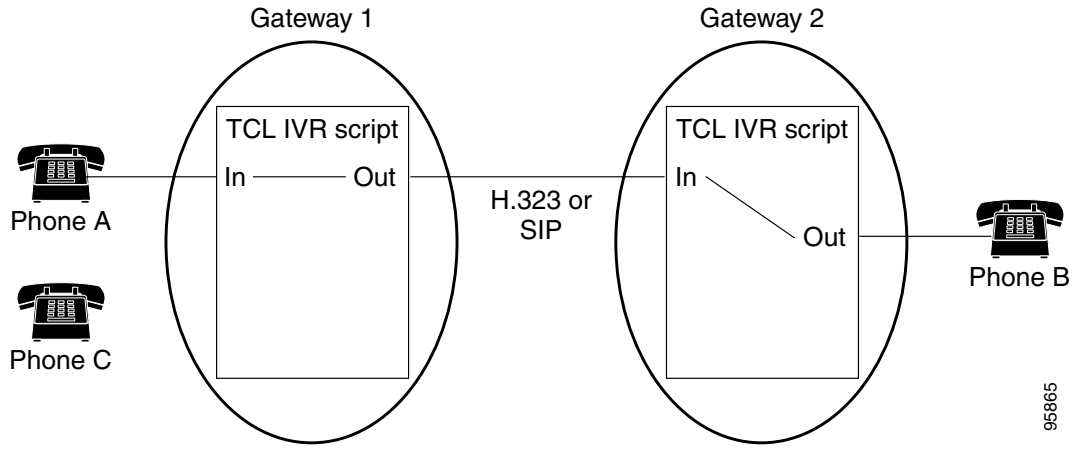
Two Gateway Scenarios with Analog Transferor

There are several call transfer scenarios that involve two Cisco IOS gateways and an analog transferor. Several of these are described in the following subsections.

XOR and XTO on Gateway 1 and XEE on Gateway 2

In the first scenario, the transferor (phone A) and transfer-target (phone C) endpoints are connected to Gateway 1. The transferee endpoint (phone B) is connected to Gateway 2. See Figure 1-9.

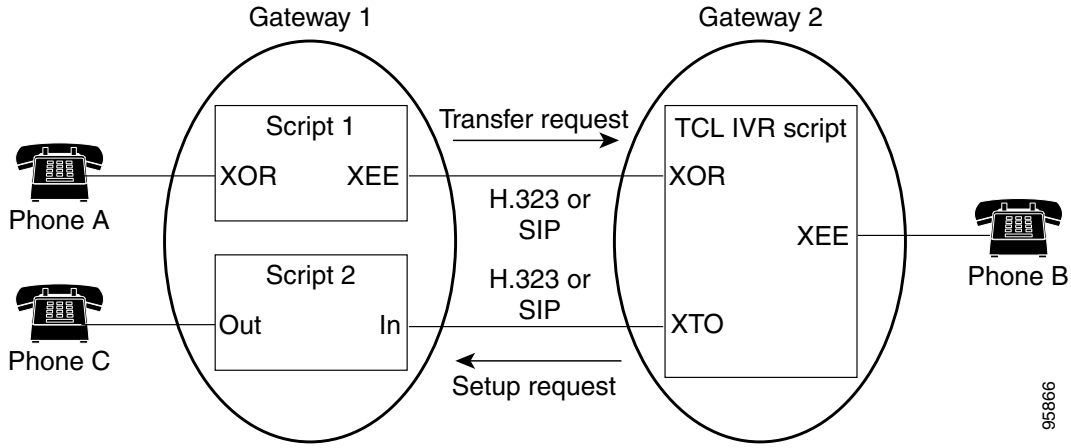
Figure 1-9 Two Gateways (XOR/XTO & XEE): Analog XOR Before Transfer



To initiate a blind transfer, the analog phone user presses hookflash, enters the transfer destination, then hangs up. The script on Gateway 1 sends a SIP or H.450 transfer request to phone B. The transfer request is received by the script handling the call between phone A and phone B on Gateway 2. This script places an outbound call to phone C and disconnects its transferor call leg when the call setup succeeds. See Figure 1-10.

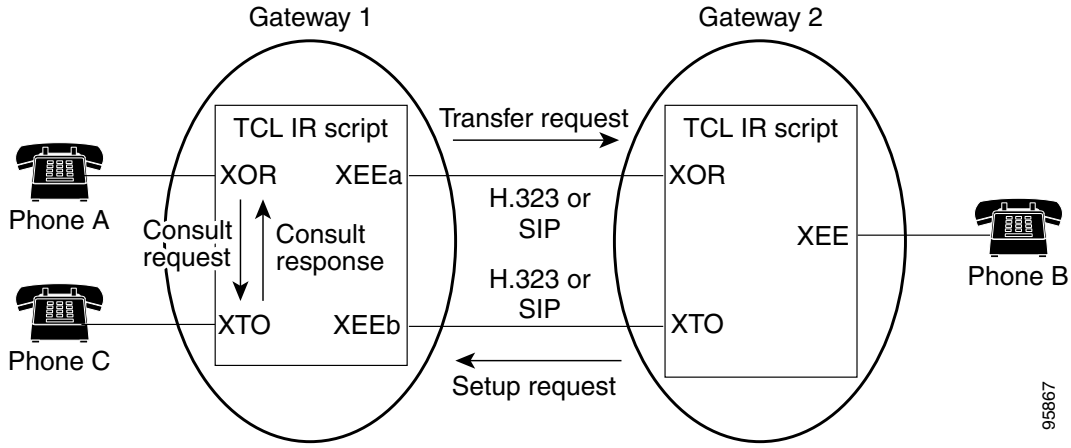
Although phone C is also connected to Gateway 1, the incoming call from phone B to phone C is handled by a separate instance of the Tcl IVR script. This script simply places a normal call to phone C, without knowledge that this call was part of a call transfer.

Figure 1-10 Two Gateways (XOR/XTO & XEE): Analog XOR Blind Transfer



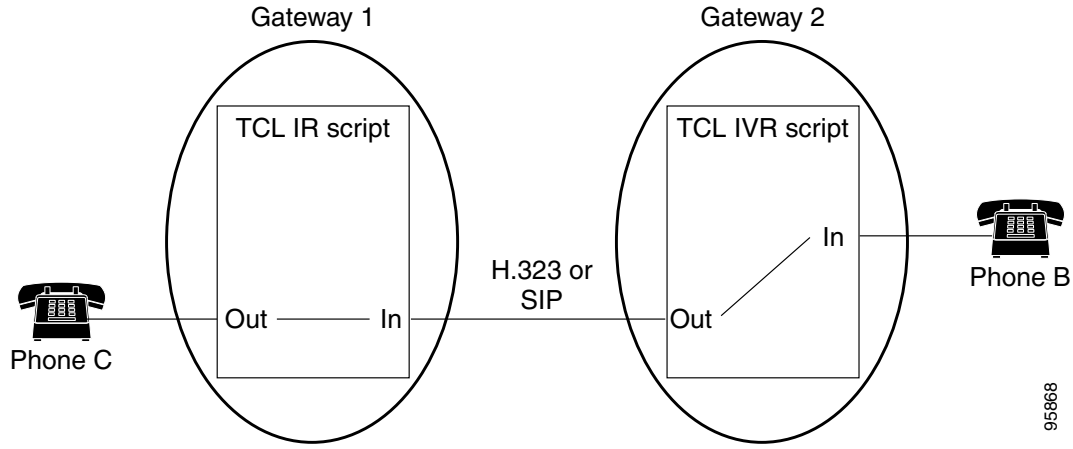
To initiate a consultation transfer, the analog phone user presses hookflash and enters the transfer destination number. The script then places a call to the transfer target so that phone A can consult with phone C. When the user commits the transfer (by hanging up), the script requests a consultation ID from the transfer target (phone C). Because phone C is a local analog phone, the gateway generates a local consultation ID and registers it to this script instance. The script then sends a SIP or H.450 transfer request to phone B that includes the consultation ID. The transfer request is received by the script handling the call on Gateway 2. This script places an outbound call to phone C and disconnects its transferor call leg when the call setup succeeds. See Figure 1-11.

Figure 1-11 Two Gateways (XOR/XTO & XEE): Analog XOR Consult Transfer



The setup request includes the consultation ID received in the transfer request. Unlike the blind transfer case above, the incoming setup request to phone C is handled by the same instance of the script that is handling the original call between phones A and B, and the consultation call between phones A and C. This script connects the incoming call to phone C and disconnects phone A. See Figure 1-12.

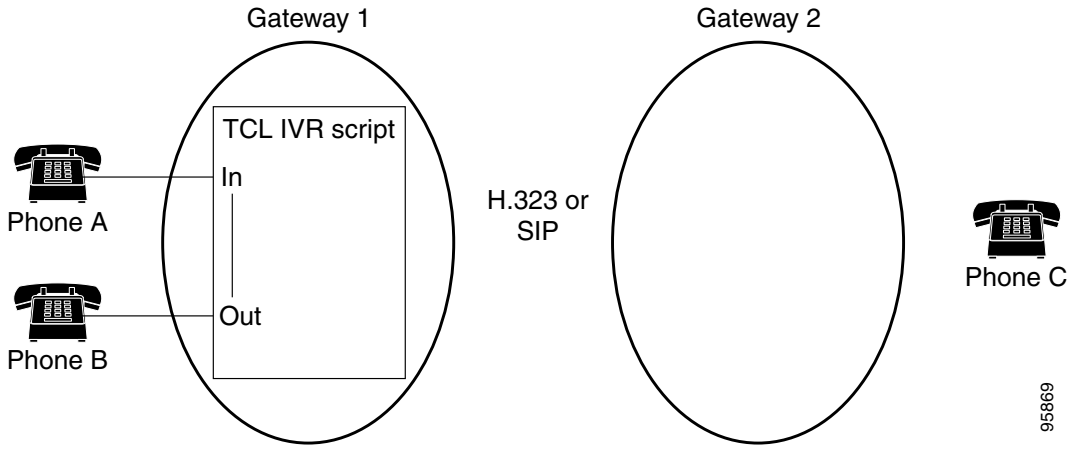
Figure 1-12 Two Gateways (XOR/XTO & XEE): Analog XOR After Transfer



XOR and XEE on Gateway 1 and XTO on Gateway 2

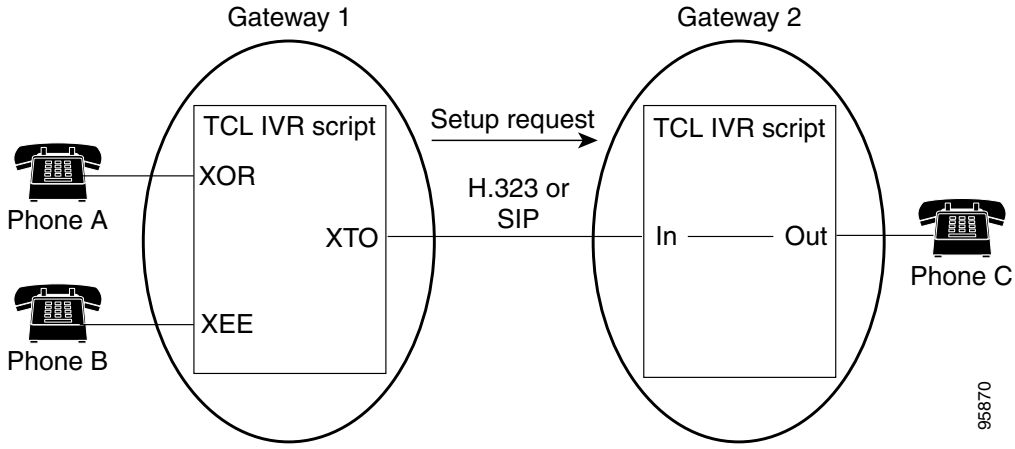
In this scenario, the transferor (phone A) and transferee (phone B) are connected to Gateway 1. The transfer target (phone C) is connected to Gateway 2. See Figure 1-13.

Figure 1-13 Two Gateways (XOR/XEE & XTO): Analog XOR Before Transfer



To initiate a blind transfer, the analog phone user presses hookflash, enters the transfer destination, and then hangs up. The script places a call to phone C by sending a SIP or H.323 setup request to Gateway 2. The script that handles this setup request on Gateway 2 places a normal call to phone C, unaware that this call was part of a call transfer. After a successful call setup, the script on Gateway 1 bridges phone B and phone C and releases the call from phone A. See Figure 1-14.

Figure 1-14 Two Gateways (XOR/XEE & XTO): Analog XOR Blind Transfer

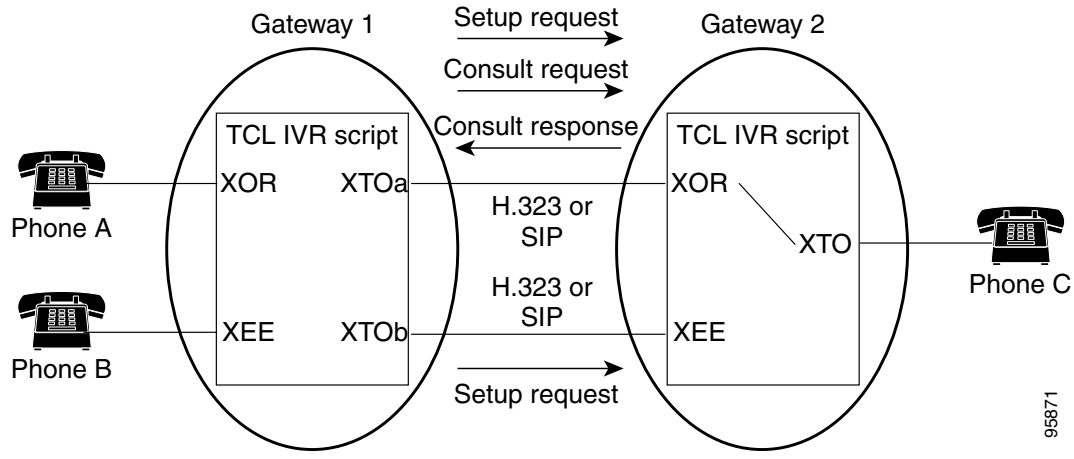


To initiate a consultation transfer, the analog phone user presses hookflash and enters the transfer destination number. The script then places a call to the transfer target so that phone A is able to consult with phone C. When the user commits the transfer (by hanging up), the script requests a consultation ID from the transfer target (phone C).

For H.450 transfers, Gateway 1 sends an H.450 consultation request message to phone C. This request is received by the script instance on Gateway 2 that is handling the call between phones A and C. This script sends a consultation response that includes a consultation ID. See Figure 1-15.

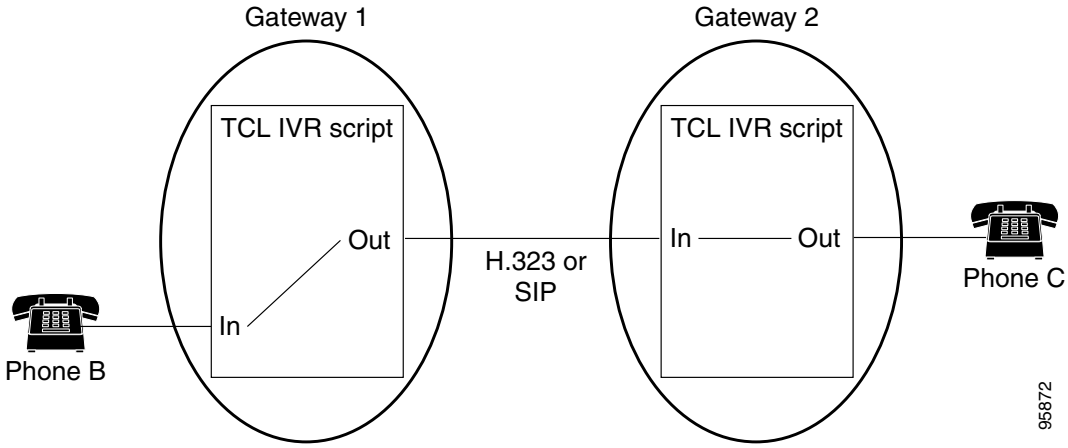
For SIP, the consultation request is not sent to phone C. Instead, a consultation ID is generated locally by Gateway 1. In both cases, when the script on Gateway 1 receives the consultation response, it sends a SIP or H.450 setup request to Gateway 2 that includes this consultation ID. When the setup request arrives at Gateway 2, it is delivered to the same script instance that is handling the consultation call between phone A and phone C.

Figure 1-15 Two Gateways (XOR/XEE & XTO): Analog XOR Consultation Transfer



This script connects the incoming call to phone C and disconnects the consultation call from phone A. See Figure 1-16.

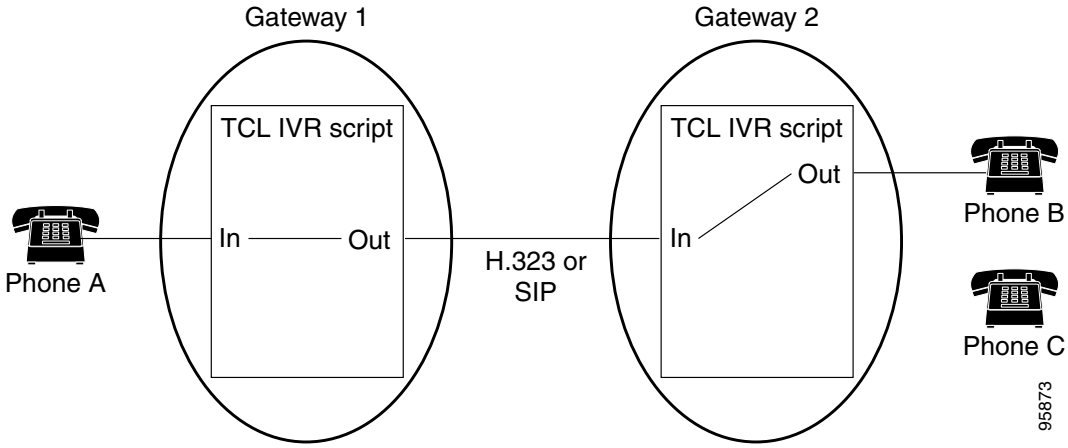
Figure 1-16 Two Gateways (XOR/XEE & XTO): Analog XOR After Transfer



XOR on Gateway 1 and XEE and XTO on Gateway 2

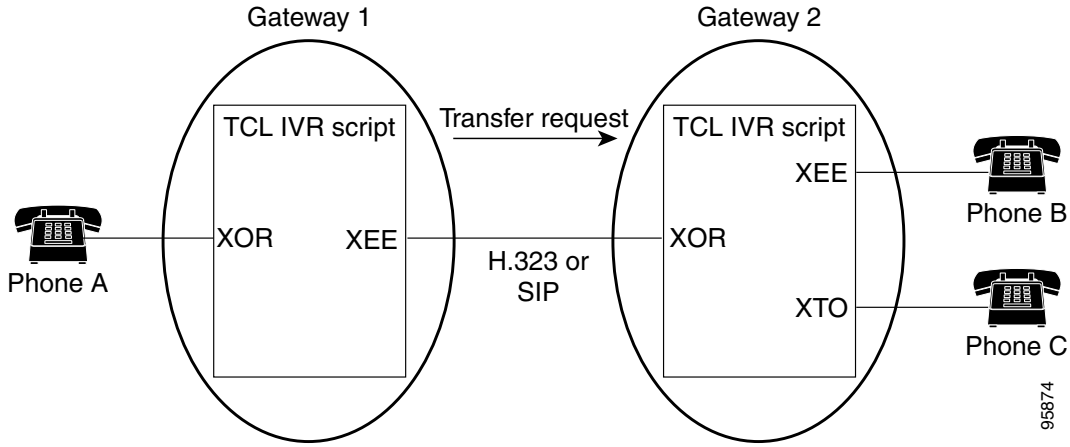
The third call transfer scenario involving two gateways is shown in Figure 1-17. The transferor (phone A) is connected to Gateway 1, and the transferee (phone B) and transfer target (phone C) are connected to Gateway 2.

Figure 1-17 Two gateways (XOR & XEE/XTO): Analog XOR Before Transfer



To initiate a blind transfer, the analog phone user presses hookflash, enters the transfer destination, then hangs up. The script on Gateway 1 sends a SIP or H.450 transfer request to phone B. The transfer request is received by the script handling the call between phone A and phone B on Gateway 2. This script places an outbound call to phone C. When the setup succeeds, this script connects phone B to phone C and disconnects the call from phone A. See Figure 1-18.

Figure 1-18 Two Gateways (XOR & XEE/XTO): Analog XOR Blind Transfer

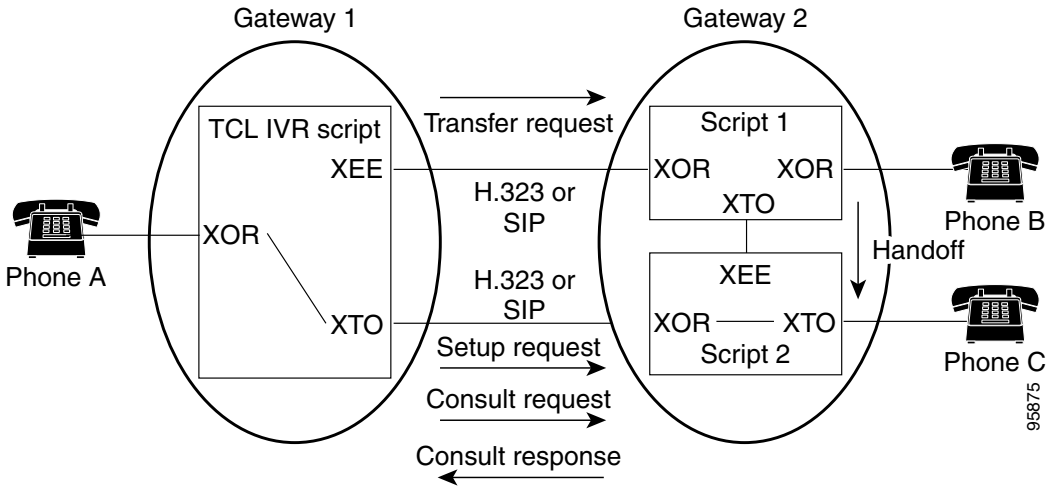


To initiate a consultation transfer, the analog phone user presses hookflash and enters the transfer destination number. The script then places a call to the transfer target so that phone A can consult with phone C. The incoming call from phone A is handled by a different script instance on Gateway 2 than is handling the call between phones A and B. See Figure 1-19.

When the user commits the transfer (by hanging up), the script on Gateway 1 requests a consultation ID from the transfer target. For H.450 transfers, Gateway 1 sends an H.450 consultation request message to phone C. The request is received by the script instance on Gateway 2 that is handling the call between phones A and C. This script sends a consultation response that includes a consultation ID.

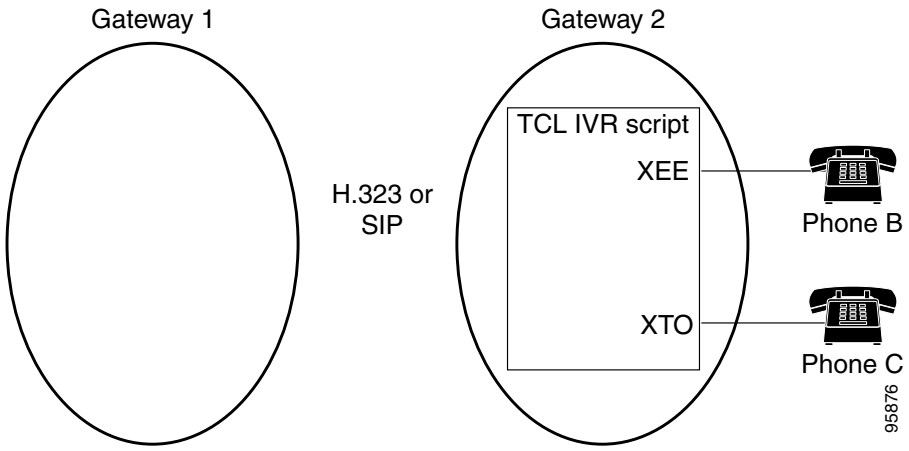
For SIP, the consultation request is not sent to phone C. Instead, a consultation ID is generated locally by Gateway 1. In both cases, when the script on Gateway 1 receives the consultation response, it sends a SIP or H.450 transfer request to Gateway 2 that includes this consultation ID.

Figure 1-19 Two Gateways (XOR & XEE/XTO): Analog XOR Consultation Transfer



The transfer request is received by the script instance handling the call between phones A and B on Gateway 2. This script places a call to phone C. The setup request includes the consultation ID received in the transfer request. Because the consultation ID included in the setup request matches the one sent to Gateway 1 in the consultation response, the call setup completes by handing off the incoming call to the second script instance. After the handoff, the original call from phone A to phone B is disconnected by the first script instance on Gateway 2 and the consultation call from phone A is disconnected by the second script instance. See Figure 1-20.

Figure 1-20 Two Gateways (XOR & XEE/XTO): Analog XOR After Transfer



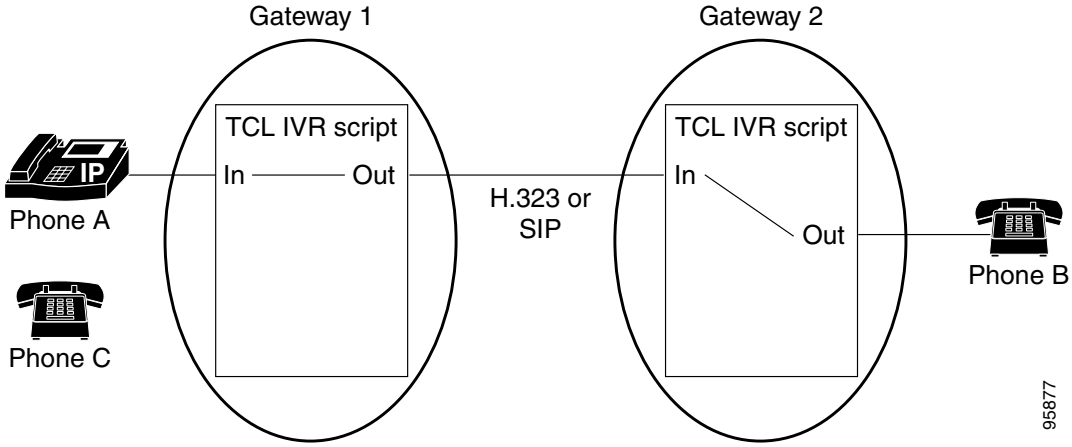
Two Gateway Scenarios with Cisco CME IP Phone Transferor

There are several call transfer scenarios that involve two Cisco IOS gateways and a Cisco CallManager Express (CME) IP phone transferor. Several of these are described in the following subsections.

XOR and XTO on Gateway 1 and XEE on Gateway 2

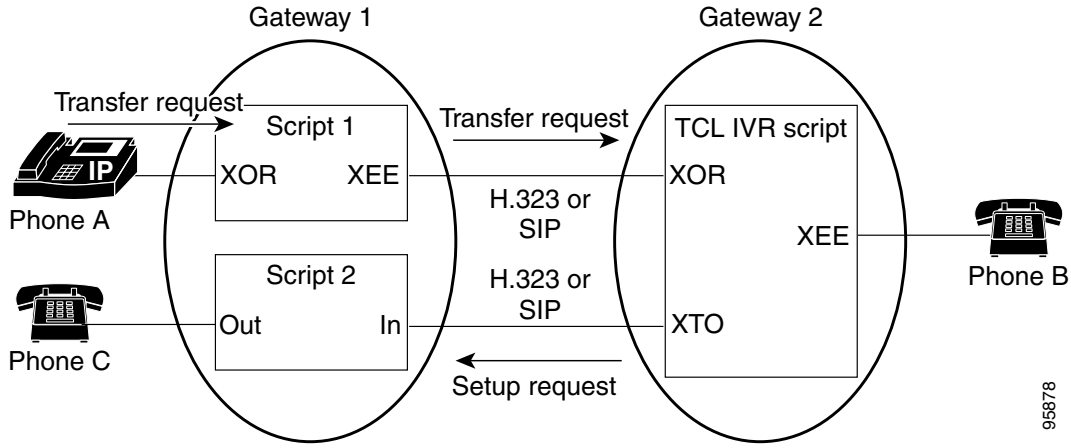
The first scenario is shown in Figure 1-21. Here, the transferor (phone A) and transfer-target (phone C) endpoints are connected to Gateway 1. The transferee endpoint (phone B) is connected to Gateway 2.

Figure 1-21 Two Gateways (XOR/XTO & XEE): Cisco CME IP Phone XOR Before Transfer



To initiate a blind transfer, the IP phone user presses the transfer button on the phone and enters the transfer destination. The script receives a transfer-request event from the phone and sends a SIP or H.450 transfer request to phone B. The transfer request is received by the script handling the call between phone A and phone B on Gateway 2. This script places an outbound call to phone C and disconnects its transferor call leg when the call setup succeeds. Although phone C is also connected to Gateway 1, the incoming call from phone B to phone C is handled by a separate instance of the Tcl IVR script. This script simply places a normal call to phone C without knowledge that this call was part of a call transfer. See Figure 1-22.

Figure 1-22 Two Gateways (XOR/XTO & XEE): Cisco CME IP Phone XOR Blind Transfer

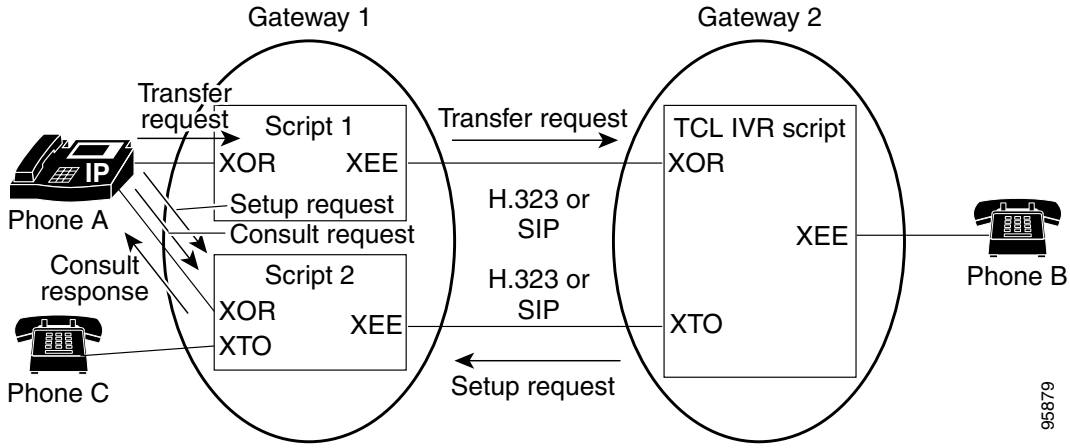


To initiate a consultation transfer, the IP phone user presses the transfer button on the phone and enters the transfer destination number. The IP phone uses a separate line to place a call to the transfer target. This call is independently handled by a second instance of the Tcl IVR script running on Gateway 1. The script instance treats the call as a normal two-party call, unaware that it is a consultation call. See Figure 1-23.

After consulting with the transfer target, the user commits the transfer by hanging up and the second script instance receives a consultation request from IP phone A. Because phone C is a local analog phone, the gateway generates a local consultation ID and registers it to this script instance. The script then sends a consultation response to IP phone A that includes this consultation ID.

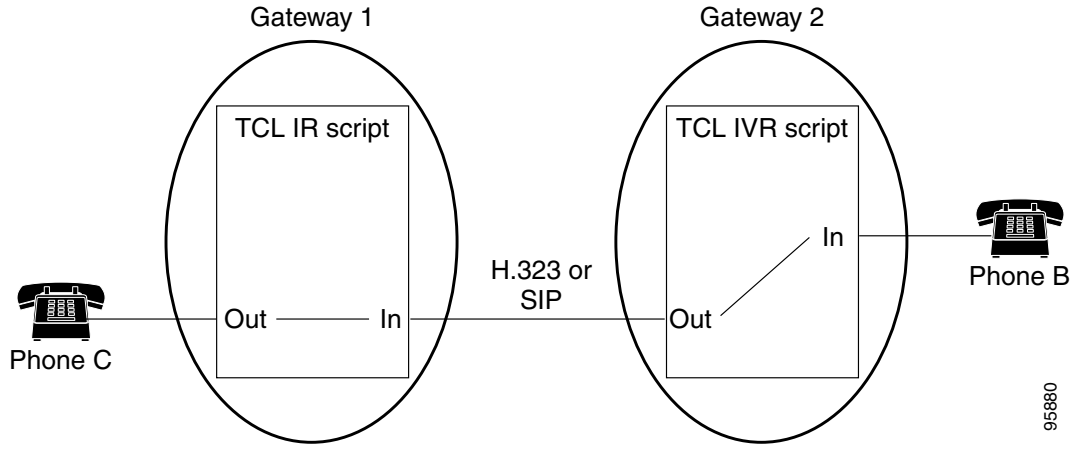
Next, the first script instance receives a transfer request from IP phone A that includes the consultation ID it received from the second script instance. This script instance then sends a SIP or H.450 transfer request to phone B that includes the consultation ID. The transfer request is received by the script handling the call between phone A and phone B on Gateway 2. This script places an outbound call to phone C and disconnects its transferor call leg when the call setup succeeds. The setup request includes the consultation ID received in the transfer request.

Figure 1-23 Two Gateways (XOR/XTO & XEE): Cisco CME IP Phone XOR Consult Transfer



The incoming setup request is delivered to the script instance on Gateway 1 that is handling the consultation call between phone A and phone C. This script connects the incoming call to phone C and releases the call from phone A. See Figure 1-24.

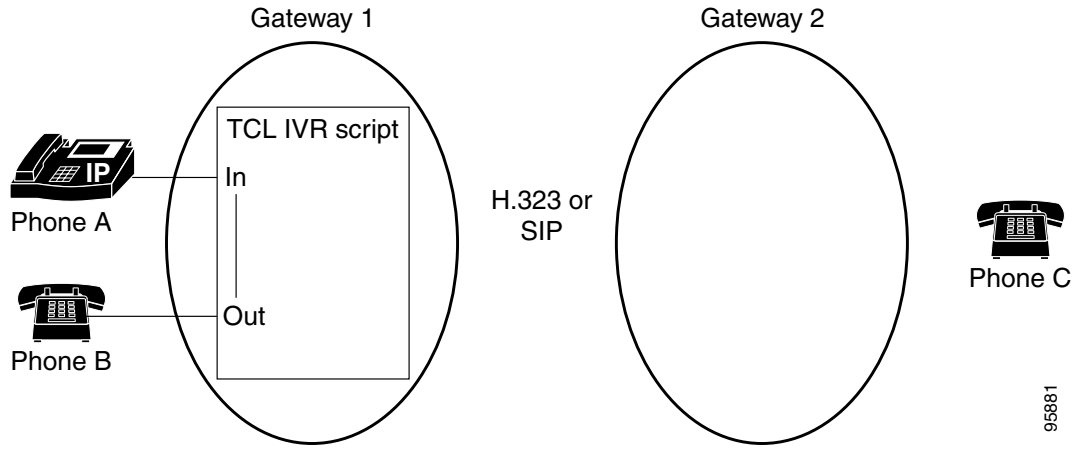
Figure 1-24 Two Gateways (XOR/XTO & XEE): Cisco CME IP Phone XOR After Transfer



XOR and XEE on Gateway 1 and XTO on Gateway 2

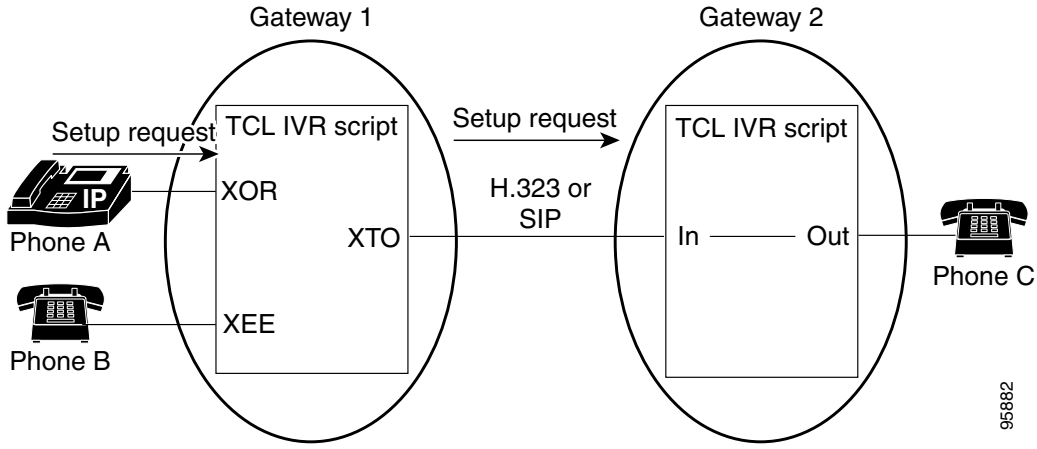
The second scenario involving two gateways and an IP phone transferor. The transferor (phone A) and transferee (phone B) are connected to Gateway 1. The transfer target (phone C) is connected to Gateway 2. See Figure 1-25.

Figure 1-25 Two Gateways (XOR/XEE & XTO): Cisco CME IP Phone XOR Before Transfer



To initiate a blind transfer, the IP phone user presses the transfer button on the phone and enters the transfer destination. The script receives a transfer request event from the phone A and places a call to phone C by sending a SIP or H.323 setup request to Gateway 2. The script that handles this setup request on Gateway 2 places a normal call to phone C, unaware that this call was part of a call transfer. After a successful call setup, the script on Gateway 1 bridges phone B and phone C and releases the call from phone A. See Figure 1-26.

Figure 1-26 Two Gateways (XOR/XEE & XTO): Cisco CME IP Phone XOR Blind Transfer



To initiate a consultation transfer, the IP phone user presses the transfer button on the phone and enters the transfer destination number. The IP phone uses a separate line to place a call to the transfer target. This call is independently handled by a second instance of the Tcl IVR script running on Gateway 1. The script instance treats this as a normal two-party call and is not aware it is a consultation call.

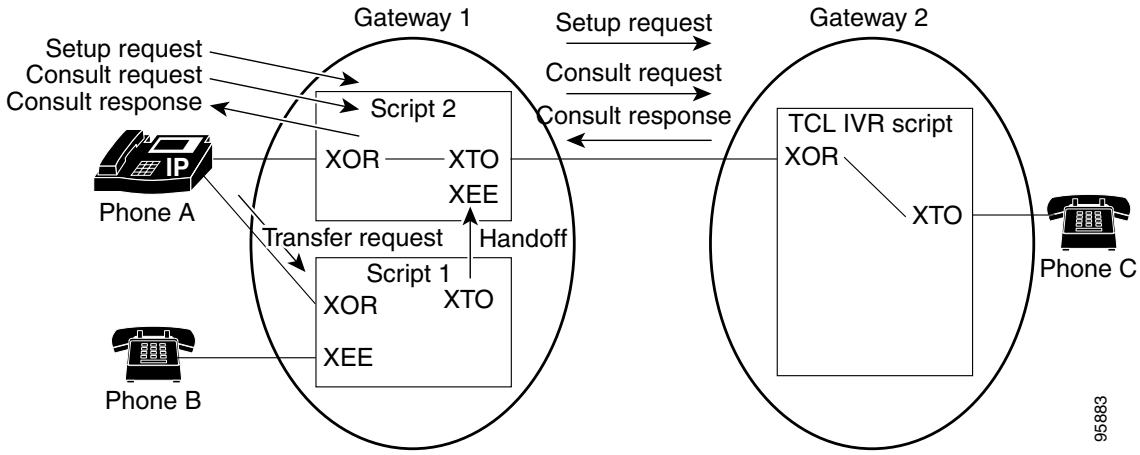
After consulting with the transfer target, the user commits the transfer by hanging up and the second script instance receives a consultation request from IP phone A. For H.450 transfers, Gateway 1 relays this consultation request to phone C by sending an H.450 consultation request message to phone C. The request is received by the script instance on Gateway 2 handling the call between phones A and C. This script sends a consultation response that includes a consultation ID. See Figure 1-27.

For SIP, the consultation request is not relayed to phone C. Instead, a consultation ID is generated locally by Gateway 1. In both cases, when the script on Gateway 1 receives the consultation response, it relays it to IP phone A. In addition, due to the internal consultation ID management scheme in the Cisco IOS application framework, the consultation ID received from Gateway 2 is registered to this script instance (the second instance).

Note Because the script instance on Gateway 2 sent a consultation response to Gateway 1, it expects to receive an incoming call from the transferee. Because the transfer was handled locally on Gateway 1 through a handoff, Gateway 2 will not receive this incoming call. A guard timer in Cisco IOS eventually expires, and the script continues processing the call between Phone A and phone C as a normal two-party call.

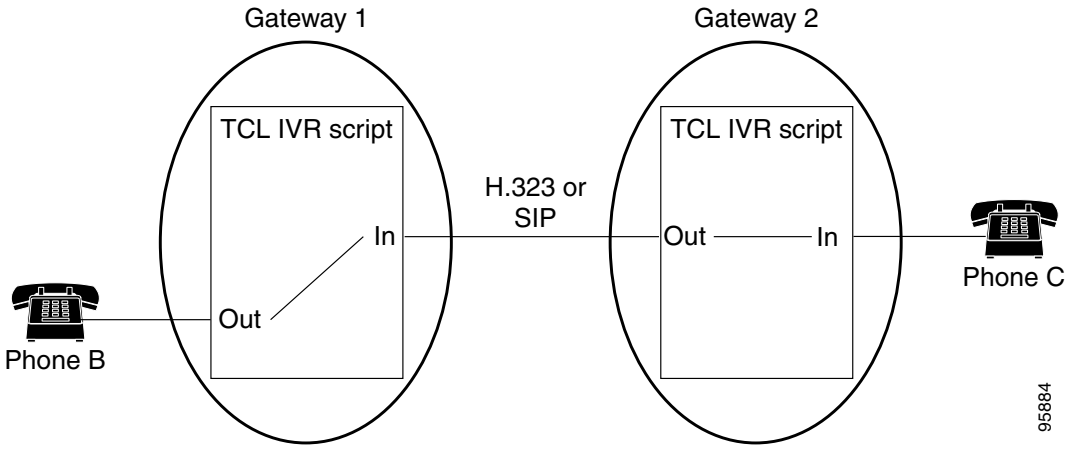
Next, the first script instance receives a transfer request from IP phone A that includes the consultation ID from the second script instance. This script instance places the outbound call to phone C that includes the consultation ID.

Figure 1-27 Two Gateways (XOR/XEE & XTO): Cisco CME IP Phone XOR Consultation Transfer



Because the consultation ID is registered to the second script instance, the transferee call leg is handed off to the second script instance. This script instance receives the handoff event and bridges the transferee and transfer target legs. The first script instance releases the transferor call leg. See Figure 1-28.

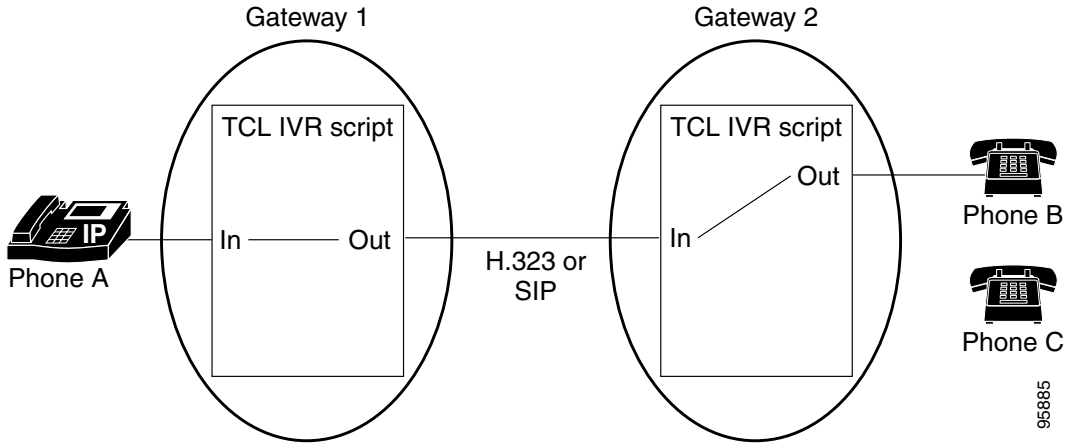
Figure 1-28 Two Gateways (XOR/XEE & XTO): Cisco CME IP Phone XOR After Transfer



XOR on Gateway 1 and XEE and XTO on Gateway 2

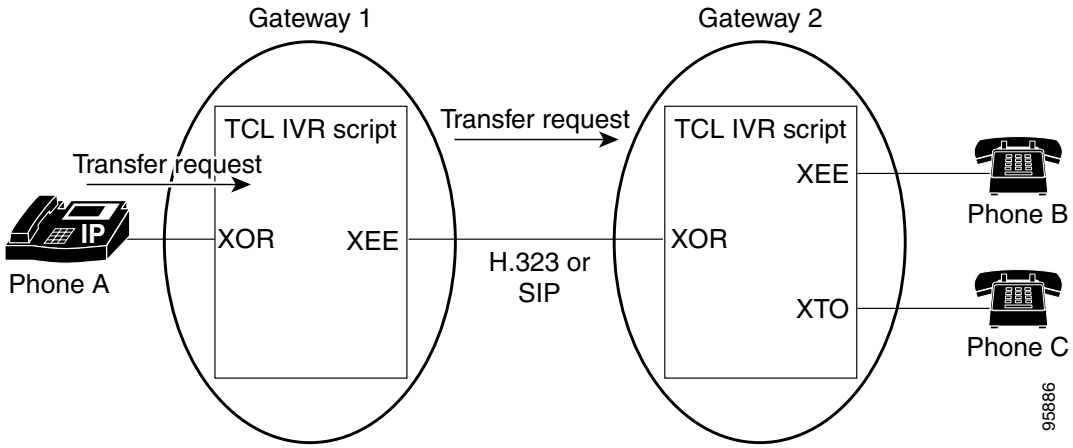
The third call transfer scenario involving two gateways and an IP phone transferor is shown in Figure 1-29. The transferor (phone A) is connected to Gateway 1, and the transferee (phone B) and transfer target (phone C) are connected to Gateway 2.

Figure 1-29 Two Gateways (XOR & XEE/XTO): Cisco CME IP Phone XOR Before Transfer



To initiate a blind transfer, the IP phone user presses the transfer button on the phone and enters the transfer destination. The script receives a transfer request event from the phone and sends a SIP or H.450 transfer request to phone B. The transfer request is received by the script handling the call between phone A and phone B on Gateway 2. This script places an outbound call to phone C. After a successful call setup, the script on Gateway 2 bridges phone B and phone C and releases the call from phone A. See Figure 1-30.

Figure 1-30 Two Gateways (XOR & XEE/XTO): Cisco CME IP Phone XOR Blind Transfer

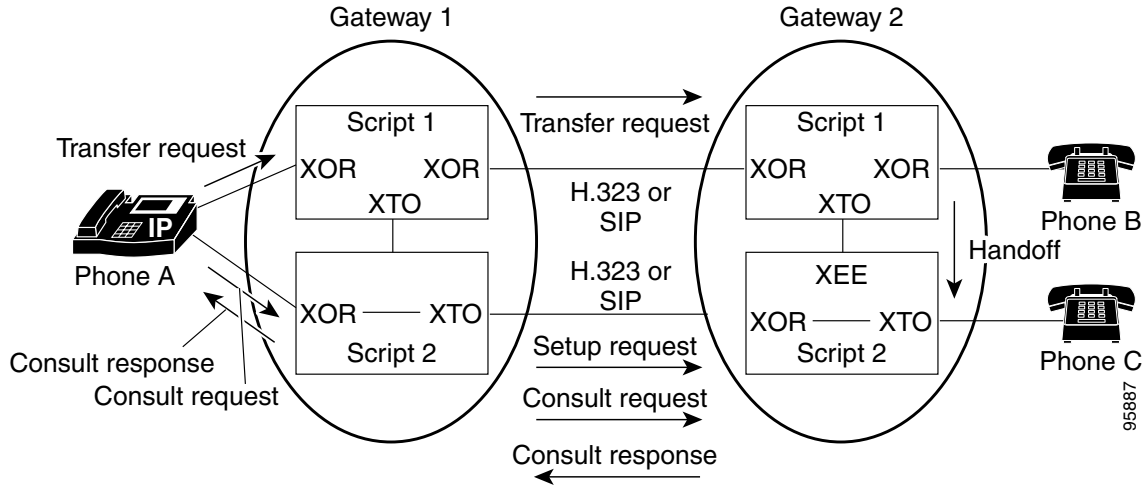


To initiate a consultation transfer, the IP phone user presses the transfer button on the phone and enters the transfer destination number. The IP phone uses a separate line to place a call to the transfer target. The call is independently handled by a second instance of the Tcl IVR script running on Gateway 1. This script instance treats the call as a normal two-party call and is not aware it is a consultation call.

After consulting with the transfer target, the user commits the transfer by hanging up and the second script instance receives a consultation request from IP phone A. For H.450 transfers, Gateway 1 relays this consultation request to phone C by sending an H.450 consultation request message to phone C. The request is received by the script instance on Gateway 2 that is handling the call between phones A and C. This script sends a consultation response that includes a consultation ID. For SIP, the consultation request is not relayed to phone C. Instead, a consultation ID is generated locally by Gateway 1. In both cases, when the script on Gateway 1 receives the consultation response, it relays it to IP phone A. In addition, due to the internal consultation ID management scheme in the Cisco IOS application framework, the consultation ID received from Gateway 2 is registered to this script instance (the second instance).

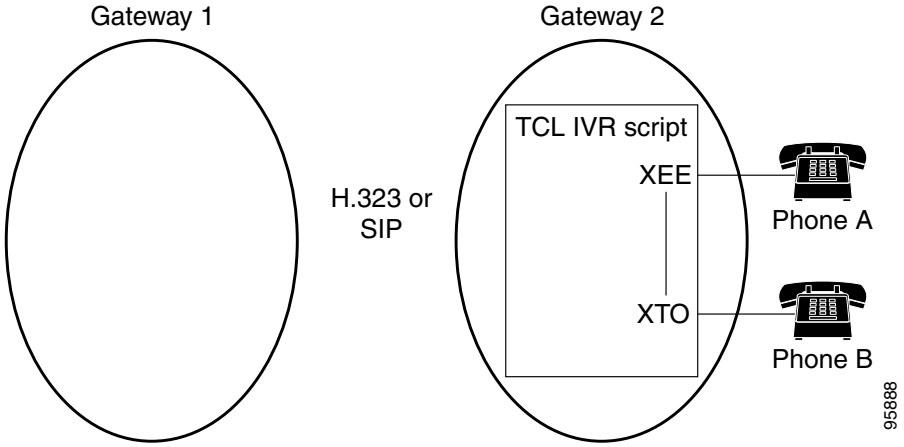
Next, the first script instance on Gateway 1 receives a transfer request from IP phone A that includes the consultation ID it received from the second script instance on Gateway 1. The script instance then sends a SIP or H.450 transfer request to phone B that includes this consultation ID. The transfer request is received by the script instance handling the call between phones A and B on Gateway 2. This script places a call to phone C. Because the consultation ID included in the setup request matches the one sent to Gateway 1 in the consultation response, the call setup is completed by handing off the incoming call to the second script instance. See Figure 1-31.

Figure 1-31 Two Gateways (XOR & XEE/XTO): Cisco CME IP Phone XOR Consultation Transfer



After the handoff, the original call from phone A to phone B is disconnected by the first script instance on Gateway 2 and the consultation call from phone A is disconnected by the second script instance. See Figure 1-32.

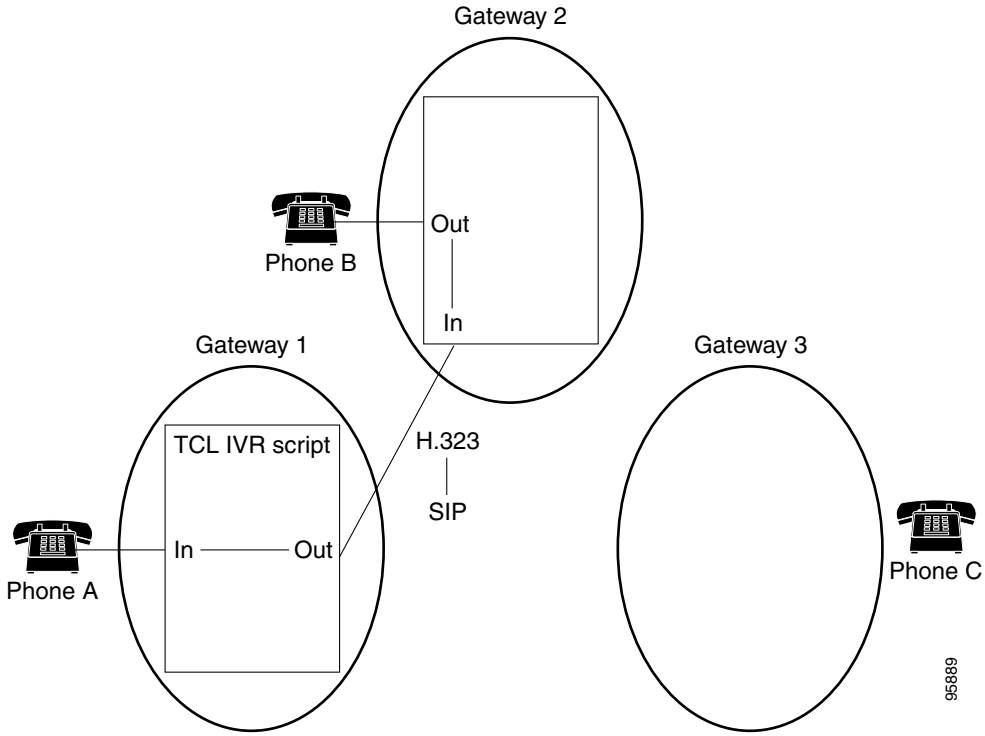
Figure 1-32 Two Gateways (XOR & XEE/XTO): Cisco CME IP Phone XOR After Transfer



Three Gateway Scenario with Analog Transferor

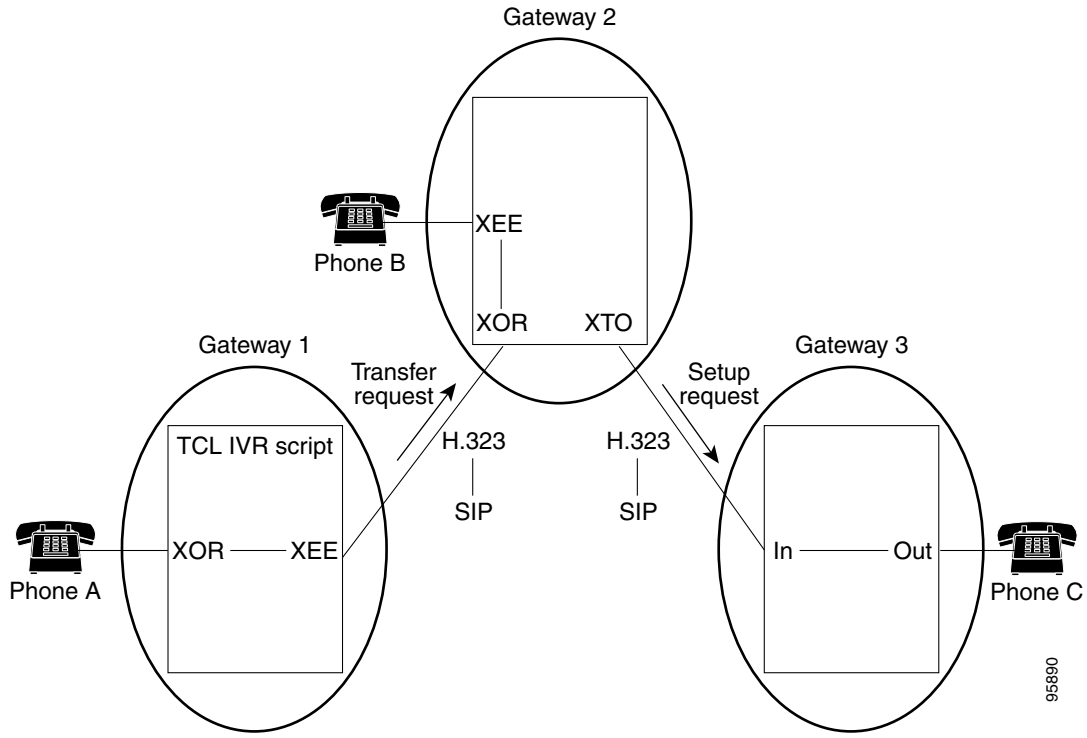
Figure 1-33 shows a scenario where three gateways are involved in the call transfer. Each call transfer participant is connected to a separate Cisco IOS gateway.

Figure 1-33 Three Gateways: Analog XOR Before Transfer



To initiate a blind transfer, the analog phone user presses hookflash, enters the transfer destination, then hangs up. The script on Gateway 1 sends a SIP or H.450 transfer request to phone B. The transfer request is received by the script handling the call on Gateway 2. This script places a regular outbound call to phone C. The script that receives the incoming call setup on Gateway 3 treats this as a normal two-party call. When the setup completes, the script on Gateway 2 sends a transfer response to phone A. The script on Gateway 1 receives the transfer response and releases the call from phone A. See Figure 1-34.

Figure 1-34 Three Gateways: Analog XOR Blind Transfer

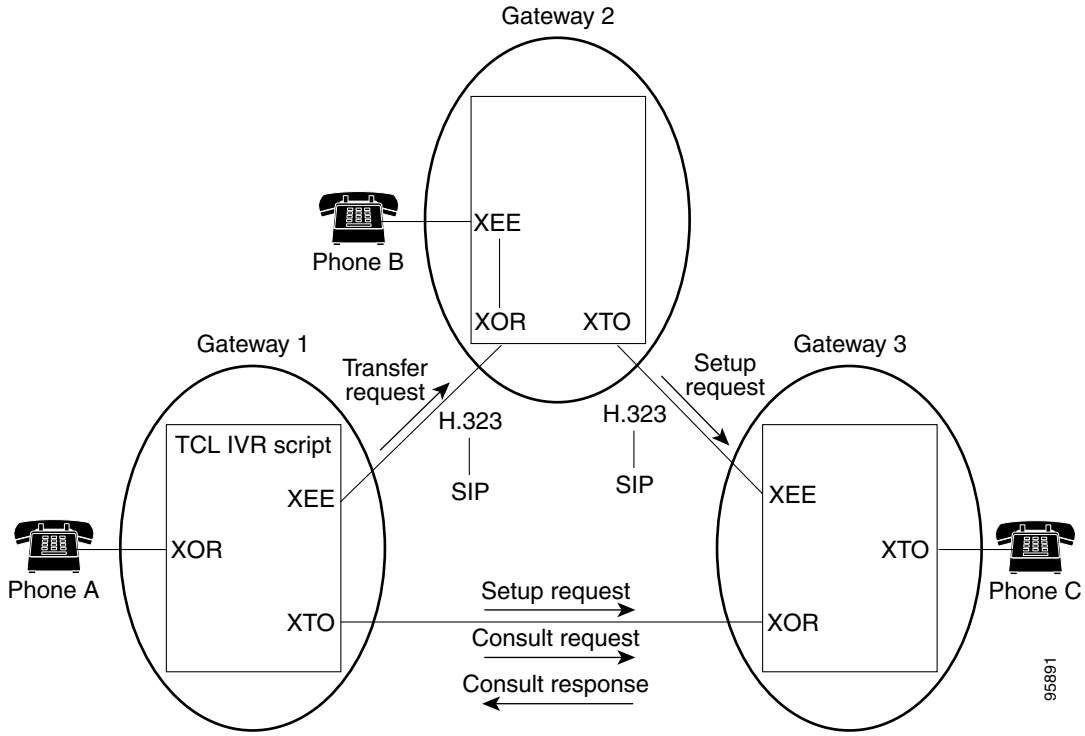


To initiate a consultation transfer, the analog phone user presses hookflash and enters the transfer destination number. The script then places a call to the transfer target so that phone A can consult with phone C. When the user commits the transfer (by hanging up), the script requests a consultation ID from the transfer target (phone C). For H.450 call transfers, a consultation request protocol message is sent to phone C. This request is received by the script instance on Gateway 3 that is handling the call between phones A and C. The script sends a consultation response that includes a consultation ID. See Figure 1-35.

For SIP, the consultation request is not sent to phone C. Instead, a consultation ID is generated locally by Gateway 1. In both cases, when the script on Gateway 1 receives the consultation response, it sends a SIP or H.450 transfer request to phone B that includes the consultation ID.

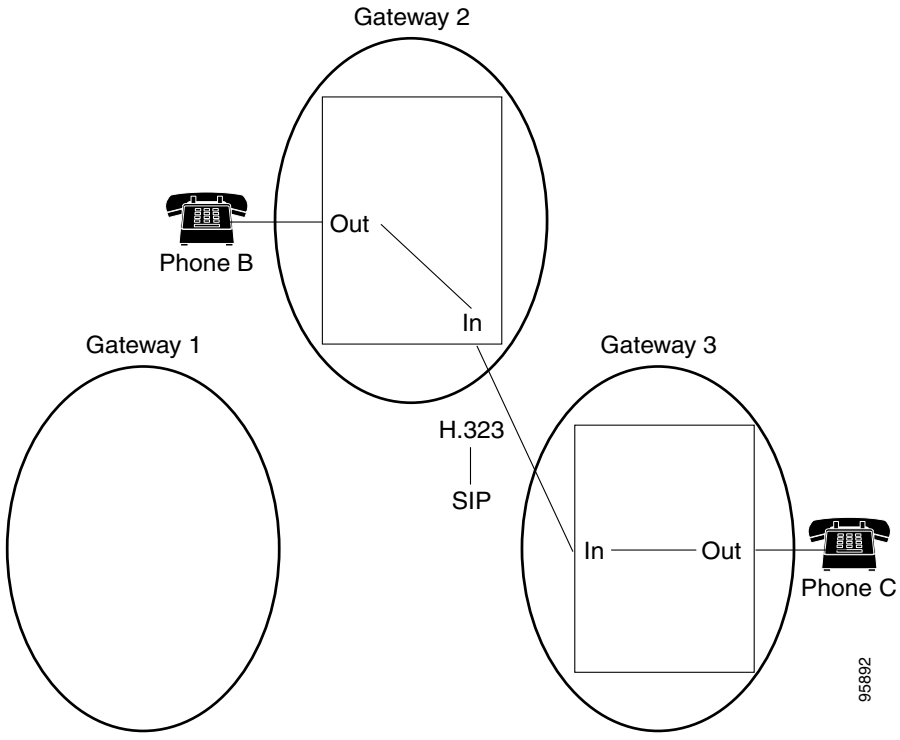
This transfer request is received by the script handling the call between phones A and B on Gateway 2. This script places a call to phone C. The setup request includes the consultation ID received in the transfer request from phone A. When the incoming setup request from phone B arrives at Gateway 2, it is delivered to the script instance handling the call between phones A and C.

Figure 1-35 Three Gateways: Analog XOR Consultation Transfer



This script instance connects the incoming call to phone C and disconnects the call from phone A. See Figure 1-36.

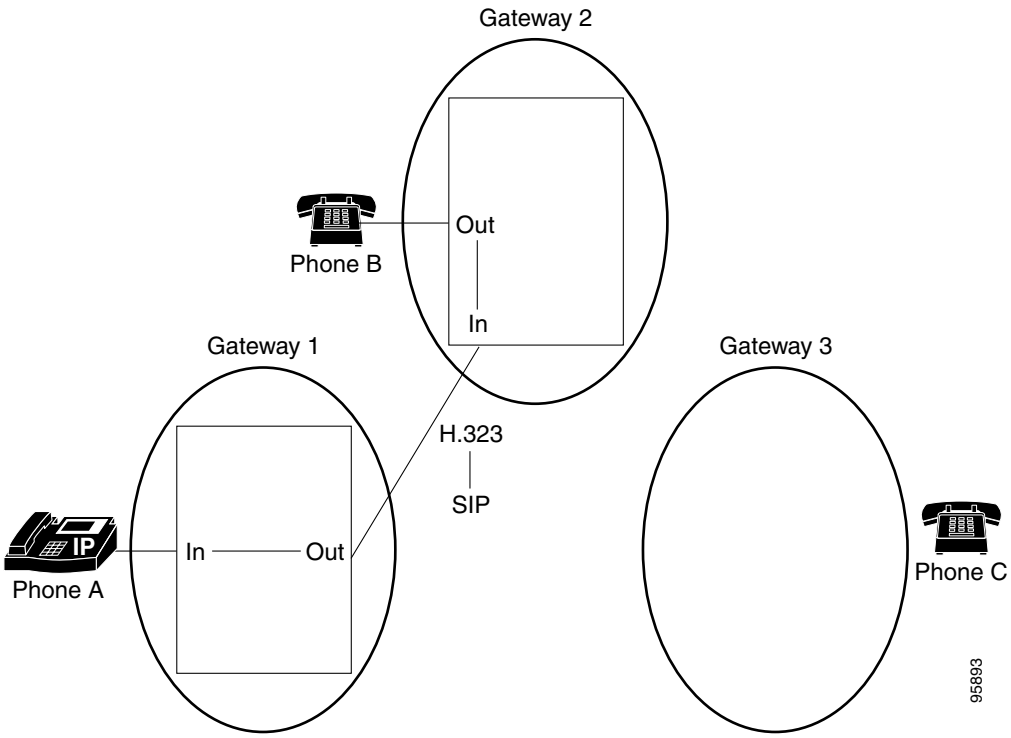
Figure 1-36 Three Gateways: Analog XOR After Transfer



Three Gateway Scenario with Cisco CME IP Phone Transferor

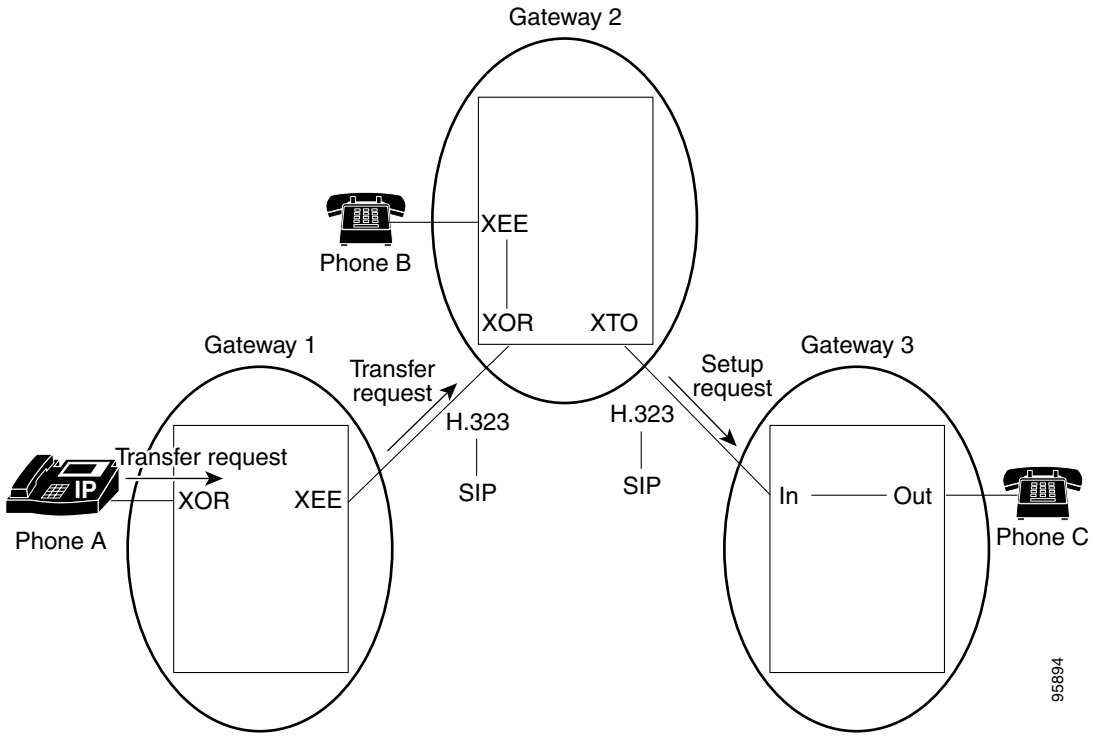
Figure 1-37 shows a scenario where three gateways are involved in the call transfer. Each call transfer participant is connected to a separate Cisco IOS gateway.

Figure 1-37 Three Gateways: Cisco CME IP Phone XOR Before Transfer



To initiate a blind transfer, the IP phone user presses the transfer button on the phone and enters the transfer destination. The script receives a transfer request event from the phone and sends a SIP or H.450 transfer request to phone B. The transfer request is received by the script handling the call on Gateway 2. This script places a regular outbound call to phone C. The script that receives the incoming call setup on Gateway 3 treats this as a normal two-party call. When the setup completes, the script on Gateway 2 sends a transfer response to phone A. The script on Gateway 1 receives the transfer response and releases the call from phone A. See Figure 1-38.

Figure 1-38 Three Gateways: Cisco CME IP Phone XOR Blind Transfer



To initiate a consultation transfer, the IP phone user presses the transfer button on the phone and enters the transfer destination number. The IP phone uses a separate line to place a call to the transfer target. This call is independently handled by a second instance of the Tcl IVR script running on Gateway 1. The script instance treats this call as a normal two-party call and is not aware it is a consultation call. See Figure 1-39.

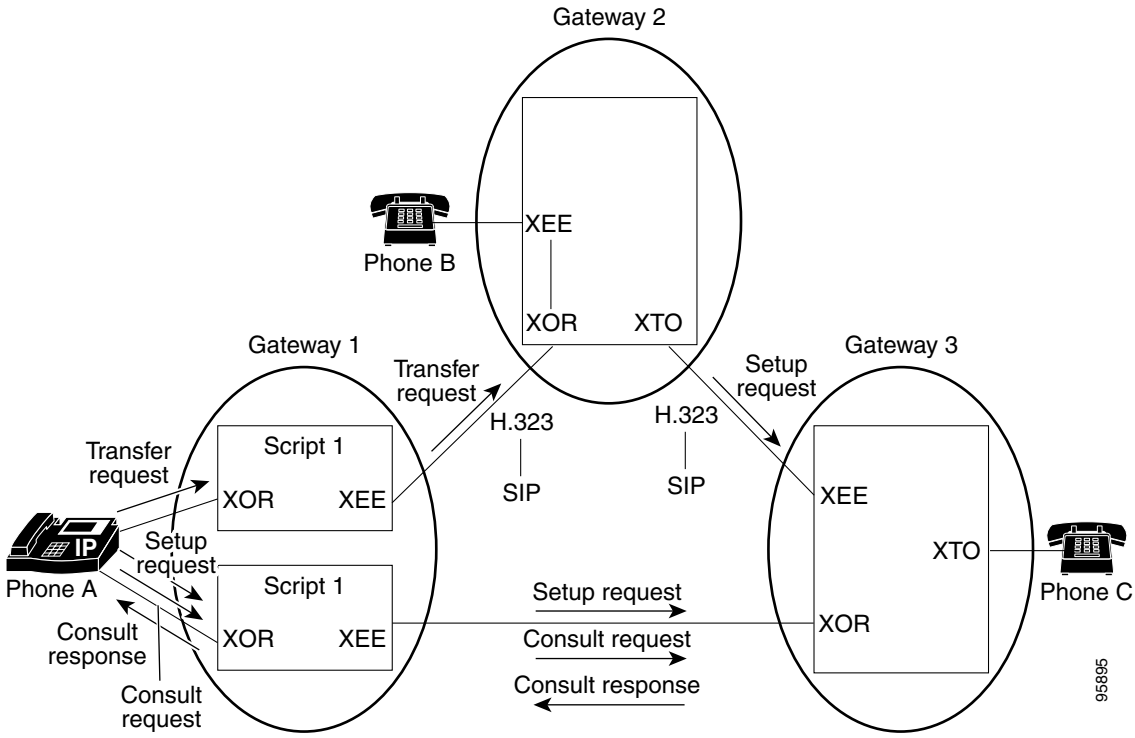
After consulting with the transfer target, the user commits the transfer by hanging up and the second script instance receives a consultation request from IP phone A. For H.450 transfers, Gateway 1 relays this consultation request to phone C by sending an H.450 consultation request message to phone C. The request is received by the script instance on Gateway 3 that is handling the call between phones A and C. This script sends a consultation response that includes a consultation ID.

For SIP, the consultation request is not relayed to phone C. Instead, a consultation ID is generated locally by Gateway 1. In both cases, when the script on Gateway 1 receives the consultation response, it relays it to IP phone A. In addition, due to the internal consultation ID management scheme in the Cisco IOS application framework, the consultation ID received from Gateway 2 is registered to this script instance (the second instance).

Next, the first script instance on Gateway 1 receives a transfer request from IP phone A that includes the consultation ID it received from the second script instance. This script instance then sends a SIP or H.450 transfer request to phone B that includes this consultation ID.

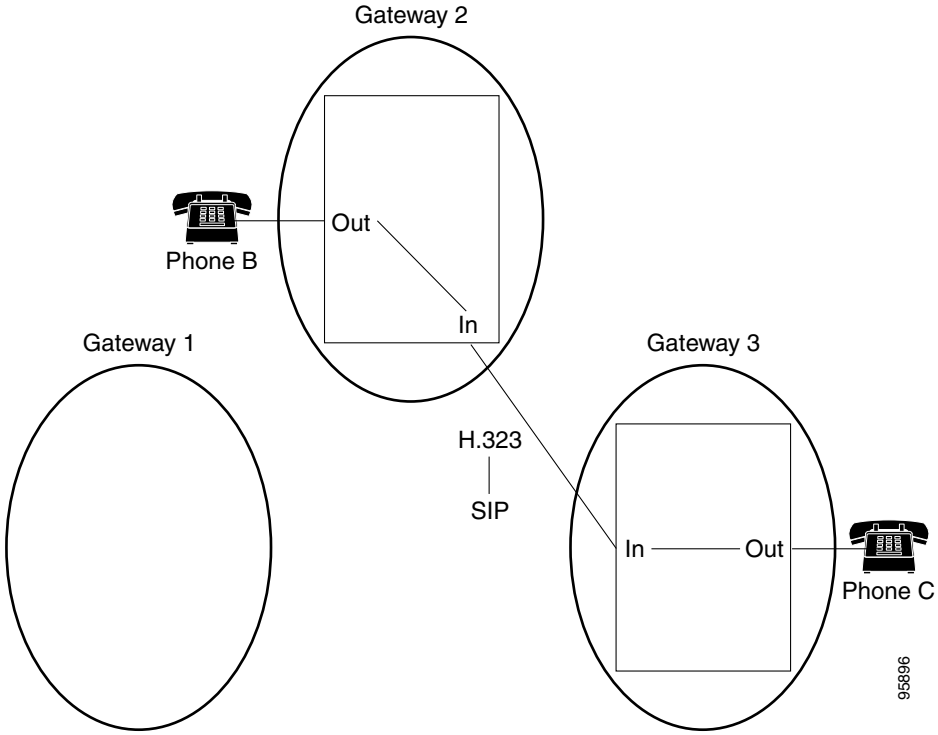
The transfer request is received by the script instance handling the call between phones A and B on Gateway 2. This script places a call to phone C. The setup request includes the consultation ID received in the transfer request from phone A. When the incoming setup request from phone B arrives at Gateway 3, it is delivered to the script instance handling the call between phones A and C.

Figure 1-39 Three Gateways: Cisco CME IP Phone XOR Consultation Transfer



This script instance connects the incoming call to phone C and disconnects the call from phone A. See Figure 1-40.

Figure 1-40 Three Gateways: Cisco CME IP Phone XOR After Transfer



Call Transfer Protocol Support

The following subsection provides an overview of the call transfer protocols supported using Tcl IVR scripting on a Cisco IOS voice gateway. Refer to the appropriate section above for various scenarios that may use these protocols.

Analog Hookflash and T1 CAS Release Link Trunk (RLT) Transfers

Transferor Support

A script cannot initiate a hookflash transfer towards a T1 CAS or analog FXO endpoint. Instead, the script can place an outbound call to the transfer target and connect the transferee and transfer target call legs after the call is established.

Transferee Support

A Tcl IVR script can receive a hookflash transfer request from a T1 CAS or analog FXS endpoint connected to the gateway. The subscriber is able to initiate a blind or consultation call transfer using hookflash and DTMF digits.

When the script receives a hookflash transfer trigger, it can provide dialtone and collect the transfer target destination through DTMF.

When the script receives a transfer commit request, it can do one of the following:

- Interwork the transfer request by propagating it to the transferee call leg. This can be done if the transferee call leg supports one of the transfer protocols described in this document.
- Place an outbound call to the transfer target and connect the transferee and transfer target call legs after the call is established.

Transfer Target Support

A Tcl IVR script cannot receive a consultation request or setup indication containing a consultation ID from an analog endpoint.

ISDN Call Transfer

Transferor Support

A Tcl IVR script can send an ISDN Two B-Channel Transfer (TBCT) request to the transferee call leg when the transferee and transfer target are both part of the same TBCT group on the PBX connected to the gateway.

When the script initiates a TBCT request, the Cisco IOS software places a call to the transfer target. When the transfer target answers, the Cisco IOS software initiates the TBCT if both the transferee and transfer target are part of the same TBCT group configured on the PBX. If the transferee and transfer target are not part of the same TBCT group, the transferee and transfer target call legs are bridged by the script. If the call is successfully transferred to the PBX, the transferee and transfer target call legs are released and the script can close the call. In some cases, the script can re-connect the transferor and transferee call legs if the transfer attempt is unsuccessful.

The script can do the following to initiate a consultation transfer:

- Place a consultation call to the transfer target device and connect the transferor and transfer target call leg when the call is established.

- If the transferee and transfer target are part of the same TBCT group, the script can do the following when the transfer is committed:
  - Request a local TBCT consultation ID.
  - Send a TBCT request to the transferee call leg. The transfer request includes the consultation ID.
  - If the call is successfully transferred to the PBX, the transferee and transfer target call legs are released, and the script can close the call.
  - In some cases, the script may re-connect the transferor and transferee call legs if the transfer attempt is unsuccessful.
- If the transferee and transfer target are not part of the same TBCT group, the transferee and transfer target call legs can be bridged by the script when the transfer is committed.

Transferee Support

A Tcl IVR script does not support any network-side ISDN call transfer protocols and is not able to receive a call-transfer request from an ISDN device.

Note It is possible to allow an ISDN subscriber to initiate a blind transfer using DTMF input to trigger the transfer. This mechanism is similar to the analog FXS and T1 CAS transfer mechanisms described above and is not discussed further in this document.

Transfer Target Support

A Tcl IVR script cannot receive a consultation request or setup indication with a consultation ID from an ISDN endpoint.

SIP Call Transfer

Transferor Support

A Tcl IVR script can send a REFER transfer request to a remote transferee call leg. The script can also initiate a consultation call when performing a consultation transfer.

The script can initiate a blind transfer by sending a REFER message to the remote transferee. If the transfer is successful, the transferee places a call the transfer target. The call is established without involvement of this script and the script can close the call. In some cases, the script can re-connect the transferor and transferee call legs if the transfer attempt is unsuccessful.

The script can do the following to initiate a consultation transfer:

- Place a consultation call to the transfer target device, and connect the transferor and transfer target call leg when the call is established.
- When the transfer is committed, request a consultation ID.

Note Unlike H.450 transfers, the script handling the consultation call between the transferor and transfer target does not receive a consultation request from the transferor. Instead, the consultation ID is generated locally by the script handling the original call between the transferor and transferee.

- Send a REFER to the transferee call leg. This includes the consultation ID. The transferee device includes the consultation ID in the INVITE message it sends to the transfer target.

- If the transfer is successful, the transferee calls the transfer target. The call is established without involvement of this script and the script can close the call.
- In some cases, the script may re-connect the transferor and transferee call legs if the transfer attempt is unsuccessful.

Transferee Support

A Tcl IVR script can receive a SIP REFER or BYE/ALSO transfer request from a remote SIP transferor. When the script receives a transfer request, the script can do one of the following:

- Interwork the transfer request by propagating it to the transferee call leg. This can be done if the transferee call leg supports one of the transfer protocols described in this document.

Note It is not currently possible to interwork SIP and H.450 transfer requests.

- Place an outbound call to the transfer target and connect the transferee and transfer target call legs after the call is established.

Transfer Target Support

When the gateway receives an INVITE request from the remote transferee that includes a consultation ID, it is delivered to the script instance handling the consultation call to the transfer target. The script can then connect the transferee and transfer target call legs and disconnect the transferor call leg.

Note Unlike H.450 transfers, the script handling the consultation call between the transferor and transfer target does not receive a consultation request from the transferor. Instead, the consultation ID is generated locally by the script that is handling the original call between the transferor and transferee.

H.450 Call Transfer

Transferor Support

A Tcl IVR script can send a H450.2 transfer request to a transferee call leg. The script can also initiate a consultation call when performing a consultation transfer.

The script can initiate a blind transfer by sending an H450.2 transfer request to the remote transferee. If the transfer is successful, the transferee calls the transfer target. The call is established without involvement of this script and the script can close the call. In some cases, the script can re-connect the transferor and transferee call legs if the transfer attempt is unsuccessful.

The script can do the following to initiate a consultation transfer:

- Place a consultation call to the transfer target device, and connect the transferor and transfer target call leg when the call is established.
- When the transfer is committed, request a consultation ID from the transfer target.
- Send an H450.2 transfer request to the transferee call leg. This includes the consultation ID received in the consultation response from the transfer target device. The transferee includes the consultation ID in the SETUP request it sends to the transfer target.
- If the transfer is successful, the transferee calls the transfer target and the call is established without involvement of this script. The script can then close the call.
- In some cases, the script can re-connect the transferor and transferee call legs if the transfer attempt is unsuccessful.

Transferee Support

A Tcl IVR script can receive an H450.2 transfer request from a remote H.323 transferor. When the script receives a transfer request, it can do one of the following:

- Interwork the transfer request by propagating it to the transferee call leg. This can be done if the transferee call leg supports one of the transfer protocols described in this document.

Note It is not possible to interwork SIP and H.450 transfer requests.

- Place an outbound call to the transfer target and connect the transferee and transfer target call legs after the call is established.

Transfer Target Support

A Tcl IVR script can receive a consultation request from a remote H450 transferor and send a consultation response that includes the consultation ID and transfer destination. This transfer destination is the number the transferee should use when placing a call to the transfer target.

When the gateway receives a SETUP request from the remote transferee that includes an H450.2 consultation ID, it is delivered to the script instance handling the consultation call to the transfer target. The script can then connect the transferee and transfer target call legs and disconnect the transferor call leg.

Cisco CallManager Express Call Transfer

Transferor Support

A Tcl IVR script cannot send a call transfer request to a local IP phone registered with the Cisco IOS gateway operating in Cisco CallManager Express (CME) mode. Instead, the script can place an outbound call to the transfer target and connect the transferee and transfer target call legs after the call is established.

Transferee Support

A Tcl IVR script can receive a call transfer request from a local IP phone registered with the Cisco IOS gateway operating in Cisco CME mode. When the script receives a transfer request, it can do one of the following:

- Interwork the transfer request by propagating it to the transferee call leg. This can be done if the transferee call leg supports one of the transfer protocols described in this document.
- Place an outbound call to the transfer target and connect the transferee and transfer target call legs after the call is established.

Transfer Target Support

A Tcl IVR script can receive a consultation request from a local Cisco CME IP phone and do one of the following:

- Interwork the consultation request by relaying it to the other call leg. This can be done if the transferee call leg supports one of the transfer protocols described in this document.
- Send a local consultation response to the IP phone that includes a locally generated consultation ID and transfer destination. This transfer destination is the number the transferee should use when placing a call to the transfer target.

When the gateway receives a SETUP request from the remote transferee that includes a consultation ID, it is delivered to the script handling the consultation call to the transfer target. The script can then connect the transferee and transfer target call legs and disconnect the transferor call leg.

SIP Subscribe and Notify

Tcl IVR 2.0 scripts provide the ability to subscribe to a SIP subscribe server and receive notify events. Applications can be invoked when notification is received, which is useful when the subscribed event will probably take a long time to complete, say several minutes or hours. In this case, the application can choose to free its instance and cause the system to create another instance to handle the notification when it is received.

The application that handles the notification need not be the same one that made the subscription. This provides the flexibility to make separate applications for handling subscriptions and notifications.

The application that made the subscription can perform any of the following tasks:

- Keep alive and handle notifications from the server.
- Free its instance and cause another instance of the same application to generate on notification.
- Free its instance and cause a different application to generate on notification.
- Make another module to handle notification.

SIP Headers

Tcl IVR 2.0 scripts can specify headers to be sent in SIP invite or H.323 setup messages. The script writer can piggy-back the header-value pairs in the destination URI after the '?". For example:

```
set destination "sip:joe@big.com?Subject=Hotel Reservation&Priority=urgent&
X-ReferenceNumber=1234567890"
leg setup destination callInfo leg_incoming
```

In cases where the destination string is an E.164 number instead of a URI, where headers cannot be appended to the destination URI, the **set** command can be used to set the headers. For example:

```
set setupSignal(Subject) "Hotel Reservation"
set setupSignal(Priority) "urgent"
set setupSignal(X-ReferenceNumber) "1234567890"
set callInfo(protoHeaders) setupSignal
set destination "4085550100"
leg setup destination callInfo leg_incoming
```

A data-passing mechanism is provided to pass application-specified headers to the SPI for outbound calls. Tcl scripts can retrieve headers using the evt_proto_headers or leg_proto_headers information tags. As of Cisco IOS release 12.3(4)T, access to headers is limited to SIP invite, subscribe, and notify messages, and to H.323 setup messages. The following list of headers, however, cannot be overwritten:

- call-ID
- Supported
- Require
- Min-SE
- Session-Expires
- Max-Forwards
- CSeq

Note Each call leg is limited to a maximum of 20K memory allocation for header passing. Each header avpair is limited to 256 characters. The application throws an error if the Tcl script tries to pass a header avpair greater than 256 characters or if the 20K memory has been used up.

If a call is handed off to an outbound application, the outbound application can retrieve all headers handed off to it from the previous application, plus headers from the incoming call leg set by the SPI, through the evt_handoff proto_headers information tag.

Application Instances

A Tcl IVR 2.0 application that is configured on the Cisco voice gateway is typically triggered by an incoming call. The application then delivers IVR services to the caller, and can create and control one or more call legs. When a voice call invokes an application, the application starts an instance, or session, of that application. The application instance executes the application script, and can place or transfer a call to a other applications. A call can initiate a single application instance or multiple application instances, depending on how the system is configured to handle the call. A single application session can manage multiple voice calls.

In Cisco IOS Release 12.3(x), you can manually start an instance of a Tcl IVR 2.0 application on the gateway without a call leg. This enables you to launch an application session on the gateway without requiring an incoming call. For example, you might write an application that monitors the status of a server group to provide a keep-alive service. An instance of this application could pass status information to other applications that are handling incoming calls. This type of service application can run on the gateway without being triggered by a call.

An instance of a Tcl IVR 2.0 application can be started on the gateway by using the **call application session start** CLI command. An application instance can communicate with other sessions on the same gateway and calls can be bridged between different sessions.

The mod_all_handles information tag can be used to retrieve a list of all the instances currently running on the gateway.

Note Tcl IVR 2.0 limits the number of subscriptions per handler to 18. Because each script instance is a handler, an application instance can only handle a maximum of 18 subscriptions simultaneously.

Session Interaction

A session is an instance of a Tcl application and is independent from other sessions in that they do not share data directly. For example, a global Tcl variable in one session is not available to another session. However, application sessions can communicate with other sessions on the gateway for the purpose of sending and receiving messages, or to hand off calls between sessions.

A Tcl session can initiate multiple outgoing call legs, and can have incoming and outgoing call legs handed off to it. A Tcl session can be running with no call legs if it was started in the CLI from a sendmsg or a notify, or if it disconnects the legs it is handling.

Session Start and Stop

The most common way for a session to start is when a Tcl application handles an incoming voice call, however, a session can also run with no call legs if it is started in the CLI, from a sendmsg, from a notification, or if it disconnects the call legs it was handling.

When a Tcl instance starts up, it receives one of the following events, depending on how it got started:

- ev_msg_indication
- ev_notify
- ev_session_indication
- ev_call_indication

If a user stops a Tcl session using the **call app session stop** CLI command, the Tcl script receives an *ev_session_terminate* event. The Tcl script is expected to close. If the Tcl script does not close after 10 seconds, the session is shut down anyway and all call legs are disconnected. This gives the Tcl script time to clean up gracefully.

If the session returns after handling an event and there are no active timers, legs, registered services, or subscriptions, the session is closed.

Sending Messages

Messages are sent to other application instances using the **sendmsg** command. The sendmsg command is an asynchronous command that does not have to wait for the destination to act on the event. If an application name is provided, a new instance of that application is generated.

Receiving Messages

Applications are notified of incoming messages from other applications through the *ev_msg_indication* event. Any parameters passed with the message are then available to the application through the evt_msg information tag. The handle of the sending application is available through the evt_msg_source information tag.

Call Handoff

In addition to passing the name of an application, the **handoff** command allows the passing of a handle. For example, assume a Tcl script gathers the caller's account number, then receives a notification that the call is being handled by another instance. The script can hand the incoming call leg to the other application instance using the handoff command, providing information in the argument string. When the other application instance returns the call leg, this application receives an *ev_returned* event.

Handoff Return

Handoff returns of a set of separate call legs received from different sessions should be done with a separate **handoff return** commands for each leg. The command "handoff return leg_all" is undefined in this case. The entire set of legs should return to the return location for the first user-defined leg.

Handoff return of a set of conferenced legs returns both legs to the same session. For example, if a session has been handed leg1 from session1 and leg2 from session2, and it conferenced the two legs together. Then the command **handoff return** *$leg2* returns both legs, conferenced together, to session2.

Service Registry

The services registry is a database that keeps track of every Tcl IVR 2.0 application instance that registers as a service. Other Tcl applications can then find and communicate with any registered application.

A Tcl session is not registered as a service through Cisco IOS software. A running instance of a Tcl IVR 2.0 application registers itself as a service by using the Tcl **service** command. The handle of any registered service can be retrieved using the mod_handle_service information tag.

Using Tcl IVR Scripts

This chapter contains information on how to create and use Tcl IVR scripts and includes the following topics:

Note Sample Tcl IVR scripts are found at http://www.cisco.com/cgi-bin/tablebuild.pl/tclware.

How Tcl IVR Version 2.0 Works

With Tcl IVR Version 2.0, scripts can be divided into three parts: the initialization procedures, the action functions, and the Finite State Machine (FSM).

- *Initialization procedures* are used to initialize variables. There are two types of initialization procedures:
  - Those functions that are called in the main code section of the script. These initialization functions are called only once—when an execution instance of the script is created. (An *execution instance* is an instance of the Tcl interpreter that is created to execute the script.) It is a good idea to initialize *global variables* (which will not change during the execution of the script) during these initialization functions. This is also a good time to read command-line interface (CLI) parameters.

  - Those functions that are called when the execution instance receives an ev_setup_indication or ev_handoff event, which mark the beginning of a call. It is good to initialize *call-specific variables* during these initialization functions.

When an execution instance of a script is created for handling a call, the execution instance is not deleted at the end of the call, but is instead held in cache. The next incoming call uses this cached execution instance, if it is available. Therefore, any global variables that were defined by the script when the first call was handled are used to handle the next call. The script should re-initialize any call-specific variables in the action function for ev_setup_indication or ev_handoff.

Variables that need to be initialized once and that will never change during the call can be initialized in the main code section of the script. For example, reading in configuration parameters is a one-time process and does not need to occur for every call. Therefore, it is more efficient to include these variables in the main code.

- *Action functions* are a set of Tcl procedures used in the definition of the FSM. These functions respond to events from the underlying system and take the appropriate actions.
- The *FSM* defines the control flow of a call by specifying the action function to call in response to a specific event under the current state.

  The starting state of the FSM is the state that the FSM is in when it receives a new call (indicated by an ev_handoff or ev_setup_indication event). This state is defined when the state machine table is registered using the **fsm define** command. From this point on, the events that are received from the system drive the state machine and the script invokes the appropriate action procedure based on the current state and the events received as defined by the **set** *variable* commands.

  The FSM supports two wildcard states and one wildcard event:

  - any_state, which can be used only as the begin state in a state transition and matches any state for which a state event combination is not already being handled.
  - same_state, which can be used only as the end state of a state transition and maintains the same state.
  - ev_any_event, which can be used to represent any event received by the script.

  For example, to create a default handler for any unhandled event, you could use:

```
set callfsm(any_state,ev_any_event)"defaultProc,same_state"
```

  To instruct the script to close a call if it receives a disconnect on any call leg, you could use:

```
set callfsm(any_state,ev_disconnected) "cleanupCall,CLOSE_CALL"
```

  In the following example, by default if the script receives an ev_disconnected event, it closes the call. However, if the script is in the media_playing state and receives an ev_disconnected event, it waits for the prompt to finish and then closes the call.

```
set callfsm(any_state,ev_disconnected) "cleanupCall,CLOSE_CALL"
set callfsm(MEDIA_PLAYING,ev_disconnected) "doSomthingProc,MEDIA_WAIT_STATE"
set callfsm(MEDIA_WAIT_STATE,ev_media_done) "cleanupCall,CLOSE_CALL"
```

  For more information about events, see Chapter 5, "Events and Status Codes."

When the gateway receives a call, the gateway hands the call to an application that is configured on the system. If the application is a Tcl script that uses Tcl IVR API Version 2.0, an execution instance of the application (or script) is created and executed.

When the script is executed, the Tcl interpreter reads the procedures in the script and executes the main section of the script (including the initialization of global variables). At this point, the **fsm define** command registers the state machine and the start state. This initialized execution instance is handed the call. From then on (until the **call close** command), when an event is received, the appropriate action procedure is called according to the current state of the call and the event received by the script.

An execution instance can handle only one call. Therefore, if the system is handling 10 calls using the same script, there will be 10 instances of that script. In between calls, the execution instances are cached to handle the next call. These cached execution instances are removed when the application is reloaded. Cached execution instances are also removed if a CLI parameter or attribute-value (AV)-pair is changed, removed, or added, or if an application is unconfigured.

Note With the previous version of the Tcl IVR API, every execution instance of a script ran under its own Cisco IOS process. As a result, handling 100 calls required 100 processes, each one running an execution instance of the script. With Tcl IVR API Version 2.0, multiple execution instances share the same Cisco IOS process. However, multiple Cisco IOS processes can be spawned to share the load—depending on the resources on the system and the number of calls.

Writing an IVR Script Using Tcl Extensions

Before you write an IVR script using Tcl, you should familiarize yourself with the Tcl extensions for IVR scripts. You can use any text editor to create your Tcl IVR script. Follow the standard conventions for creating a Tcl script and incorporate the Tcl IVR commands as necessary.

A sample script is provided in this section to illustrate how the Tcl IVR API Version 2.0 commands can be used.

Note If the caller hangs up, the script stops running and the call legs are cleared. No further processing is done by the script.

Prompts in Tcl IVR Scripts

Tcl IVR API Version 2.0 allows two types of prompts: memory-based and RTSP-based prompts.

- With memory-based prompts, the prompt (audio file) is read into memory and then played out to the appropriate call legs as needed. Memory-based prompts can be read from Flash memory, a TFTP server, or an FTP server.
- With RTSP-based prompts, you can use an external (RTSP-capable) server to play a specific audio file or content and to stream the audio to the appropriate call leg as needed. Some platforms may not support RTSP-based prompts. For those platforms, the prompt fails with a status code in the ev_media_done event.

As mentioned earlier, through the use of dynamic prompts, Tcl IVR API Version 2.0 also provides some basic TTS functionality, like playing numbers, dollar amounts, date, and time. It also allows you to classify prompts using different languages so that when the script is instructed to play a particular prompt, it automatically plays the prompt in the active or specified language.

Note When setting up scripts, we recommend not to using RTSP with very short prompts or dynamic prompts because of poor performance.

Sample Tcl IVR Script

The following example shows how to use the Tcl IVR API Version 2.0 commands. We recommend that you start with the header information. This includes the name of the script, the date that the script was created and by whom, and some general information about what the script does.

We also recommend that you include a version number for the script, using a three-digit system, where the first digit indicates a major version of the script, the second digit is incremented with each minor revision (such as a change in function within the script), and the third digit is incremented each time any other changes are made to the script.

The following sample script plays dial-tone, collects digits to match a dial-plan, places an outgoing call to the destination, conferences the two call legs, and destroys the conference call legs and the disconnect call legs, when anyone hangs up.

```
# app_session.tcl
# Script Version 1.0.1
#------------------------------------------------------------------
# August 1999, Saravanan Shanmugham
#
# Copyright (c) 1998, 1999 by cisco Systems, Inc.
# All rights reserved.
#------------------------------------------------------------------
#
# This tcl script mimics the default SESSION app
#
#
# If DID is configured, place the call to the dnis.
# Otherwise, output dial-tone and collect digits from the
# caller against the dial-plan.
#
# Then place the call. If successful, connect it up. Otherwise,
# the caller should hear a busy or congested signal.

# The main routine establishes the state machine and then exits.
# From then on, the system drives the state machine depending on the
# events it receives and calls the appropriate Tcl procedure.
```

Next, we define a series of procedures.

The **init** procedure defines the initial parameters of the digit collection. In this procedure:

- Users are allowed to enter information before the prompt message is complete.
- Users are allowed to abort the process by pressing the asterisk key.
- Users must indicate that they have completed their entry by pressing the pound key.

```
proc init { } {
    global param

    set param(interruptPrompt) true
    set param(abortKey) *
    set param(terminationKey) #

}
```

The **act_Setup** procedure is executed when an ev_setup_indication event is received. It gathers the information necessary to place the call. In this procedure:

- A setup acknowledgement is sent to the incoming call leg.

- If the call is Direct Inward Dial (DID), the destination is set to the Dialed Number Information Service (DNIS), and the system responds with a proceeding message on the incoming leg and tries to set up the outbound leg with the **leg setup** command.
- If not, a dial tone is played on the incoming call leg and digits are collected against a dial plan.

```
proc act_Setup { } {
    global dest
    global beep

    set beep 0
    leg setupack leg_incoming

    if { [infotag get leg_isdid] } {
        set dest [infotag get leg_dnis]
        leg proceeding leg_incoming
        leg setup $dest callInfo leg_incoming
        fsm setstate PLACECALL
    } else {

        playtone leg_incoming tn_dial

        set param(dialPlan) true
        leg collectdigits leg_incoming param
      }

}
```

The **act_GotDest** procedure is executed when an ev_collectdigits_done event is received. It determines whether the collected digits match the dial plan, in which case the call should be placed. In this procedure:

- If the digit collection succeeds with a match to the dial plan (cd_004), the script proceeds with setting up the call.
- Otherwise, the script reports the error and ends the call. For a list of other digit collection status values, see the "Digit Collection Status" section on page 5-7.

```
proc act_GotDest { } {
    global dest

    set status [infotag get evt_status]

    if {  $status == "cd_004" } {
        set dest [infotag get evt_dcdigits]
        leg proceeding leg_incoming
        leg setup $dest callInfo leg_incoming

    } else {
        puts "\nCall [infotag get con_all] got event $status collecting destination"
        call close
    }
}
```

The **act_CallSetupDone** procedure is executed when an ev_setup_done event is received. It determines whether there is a time limit on the call. In this procedure:

- When the call is successful (ls_000), the script obtains the amount of credit time.
- If a value other than unlimited or uninitialized is returned, a timer is started.
- If the call is not successful, the script reports the error and closes the call. For a list of other leg setup status values, see the "Leg Setup Status" section on page 5-10.

```
proc act_CallSetupDone { } {
    global beep
```

```
    set status [infotag get evt_status]

    if { $status == "ls_000"} {

        set  creditTimeLeft [infotag get leg_settlement_time leg_outgoing]

        if { ($creditTimeLeft == "unlimited") ||
             ($creditTimeLeft == "uninitialized") } {
            puts "\n Unlimited Time"
        } else {
            # start the timer for ...
            if { $creditTimeLeft < 10 } {
                set beep 1
                set delay $creditTimeLeft
            } else {
                set delay [expr $creditTimeLeft - 10]
            }
            timer start leg_timer $delay leg_incoming
        }
    } else {
        puts "Call [infotag get con_all] got event $status while placing an outgoing
call"
        call close
    }
}
```

The **act_Timer** procedure is executed when an ev_leg_timer event is received. It is used in the last 10 seconds of credit time and warns the user that time is expiring and terminates the call when the credit limit is reached. In this procedure:

- While there is time left, the script inserts a beep to warn the user that time is running out.
- Otherwise, the “out of time” audio file is played and the state machine is instructed to disconnect the call.

```
proc act_Timer { } {
    global beep
    global incoming
    global outgoing

    set incoming [infotag get leg_incoming]
    set outgoing [infotag get leg_outgoing]

    if { $beep == 0 } {
        #insert a beep ...to the caller
        connection destroy con_all
        set beep 1
    } else {
        media play leg_incoming flash:out_of_time.au
        fsm setstate CALLDISCONNECTED
    }
}
```

The **act_Destroy** procedure is executed when an ev_destroy_done event is received. It plays a beep to the incoming call leg.

```
proc act_Destroy { } {
    media play leg_incoming flash:beep.au
}
```

The **act_Beeped** procedure is executed when an ev_media_done event is received. It creates a connection between the incoming and outgoing call legs.

```
proc act_Beeped { } {
    global incoming
    global outgoing

    connection create $incoming $outgoing
}
```

The **act_ConnectedAgain** procedure is executed when an ev_create_done event is received. It resets the timer on the incoming call leg to 10 seconds.

```
proc act_ConnectedAgain { } {
    timer start leg_timer 10 leg_incoming
}
```

The **act_Ignore** procedure reports "Event Capture."

```
proc act_Ignore { } {
# Dummy
    puts "Event Capture"
}
```

The **act_Cleanup** procedure is executed when an ev_disconnected event is received and when the state is CALLDISCONNECTED. It closes the call.

Note When the script receives an ev_disconnected event, the script has 15 seconds to clear the leg with the **leg disconnect** command. After 15 seconds, a timer expires, the script is cleaned up, and an error message is displayed to the console. This avoids the situation where a script might not have cleared a leg after a disconnect.

```
proc act_Cleanup { } {
    call close
}
```

Finally, we put all the procedures together in a main routine. The main routine defines a Tcl array that defines the actual state transitions for the various state and event combinations. It registers the state machine that will drive the calls. In the main routine:

- If the call is disconnected while in any state, the **act_Cleanup** procedure is called and the state remains as it was.
- If a "setup indication" event is received while in the CALL_INIT state, the **act_Setup** procedure is called (to gather the information necessary to place the call) and the state is set to GETDEST.
- If a "digit collection done" event is received while in the GETDEST state, the **act_GotDest** procedure is called (to determine whether the collected digits match the dial plan and the call can be placed) and the state is set to PLACECALL.
- If a "setup done" event is received while in the PLACECALL state, the **act_CallSetupDone** procedure is called (to determine whether there is a time limit on the call) and the state is set to CALLACTIVE.
- If a "leg timer" event is received while in the CALLACTIVE state, the **act_Timer** procedure is called (to warn the user that time is running out) and the state is set to INSERTBEEP.
- If a "destroy done" event is received while in the INSERTBEEP state, the **act_Destroy** procedure is called (to play a beep on the incoming call leg) and the state remains INSERTBEEP.
- If a "media done" event is received while in the INSERTBEEP state, the **act_Beeped** procedure is called (to reconnect the incoming and outgoing call legs) and the state remains INSERTBEEP.

- If a “create done” event is received while in the INSERTBEEP state, the **act_ConnectedAgain** procedure is called (to reset the leg timer on the incoming call leg to 10 seconds) and the state is set to CALLACTIVE.
- If a “disconnect” event is received while in the CALLACTIVE state, the **act_Cleanup** procedure is called (to end the call) and the state is set to CALLDISCONNECTED.
- If a “disconnect” event is received while in the CALLDISCONNECTED state, the **act_Cleanup** procedure is called (to end the call) and the state remains CALLDISCONNECTED.
- If a “media done” event is received while in the CALLDISCONNECTED state, the **act_Cleanup** procedure is called (to end the call) and the state remains CALLDISCONNECTED.
- If a “disconnect done” event is received while in the CALLDISCONNECTED state, the **act_Cleanup** procedure is called (to end the call) and the state remains CALLDISCONNECTED.
- If a “leg timer” event is received while in the CALLDISCONNECTED state, the **act_Cleanup** procedure is called (to end the call) and the state remains CALLDISCONNECTED.

```
init

#---------------------------------
#   State Machine
#---------------------------------
  set TopFSM(any_state,ev_disconnected) "act_Cleanup,same_state"
  set TopFSM(CALL_INIT,ev_setup_indication) "act_Setup,GETDEST"
  set TopFSM(GETDEST,ev_collectdigits_done) "act_GotDest,PLACECALL"
  set TopFSM(PLACECALL,ev_setup_done)  "act_CallSetupDone,CALLACTIVE"
  set TopFSM(CALLACTIVE,ev_leg_timer) "act_Timer,INSERTBEEP"
  set TopFSM(INSERTBEEP,ev_destroy_done) "act_Destroy,same_state"
  set TopFSM(INSERTBEEP,ev_media_done)  "act_Beeped,same_state"
  set TopFSM(INSERTBEEP,ev_create_done)    "act_ConnectedAgain,CALLACTIVE"
  set TopFSM(CALLACTIVE,ev_disconnected)   "act_Cleanup,CALLDISCONNECTED"
  set TopFSM(CALLDISCONNECTED,ev_disconnected) "act_Cleanup,same_state"
  set TopFSM(CALLDISCONNECTED,ev_media_done)  "act_Cleanup,same_state"
  set TopFSM(CALLDISCONNECTED,ev_disconnect_done) "act_Cleanup,same_state"
  set TopFSM(CALLDISCONNECTED,ev_leg_timer) "act_Cleanup,same_state"
```

Initialization and Setup of State Machine

The following command is used to initialize and set up the State Machine (SM):

```
fsm define TopFSM  CALL_INIT
```

Testing and Debugging Your Script

It is important to thoroughly test a script before it is deployed. To test a script, you must place it on a router and place a call to activate the script. When you test your script, make sure that you test every procedure in the script and all variations within each procedure.

You can view debugging information applicable to the Tcl IVR scripts that are running on the router. The **debug voip ivr** command allows you to specify the type of debug output you want to view. To view debug output, enter the following command in privileged-exec mode:

[**no**] **debug voip ivr** [**states** | **error** | **tclcommands** | **callsetup** | **digitcollect** | **script** | **dynamic** | **applib** | **settlement** | **all**]

For more information about the **debug voip ivr** command, refer to the *Interactive Voice Response Version 2.0 on Cisco VoIP Gateways* document.

The output of any Tcl **puts** commands is displayed if script debugging is on.

Possible sources of errors are:

- An unknown or misspelled command (for example, if you misspell media play as mediaplay)
- A syntax error (such as, specifying an invalid number of arguments)
- Executing a command in an invalid state (for example, executing the **media pause** command when no prompt is playing)
- Using an information tag (info-tag) in an invalid scope (for example, specifying evt_dcdigits when not handling the ev_collectdigits_done event). For more information about info-tags, see Chapter 4, "Information Tags."

In most cases, an error such as these causes the underlying infrastructure to disconnect the call legs and clean up.

Loading Your Script

To associate an application with your Tcl IVR script, use the following command:

```
(config)# call application voice application_name script_url
```

After you associate an application with your Tcl IVR script, use the following command to configure parameters:

```
(config)# call application voice application_name script_url [parameter value]
```

In this command:

- *application_name* specifies the name of the Tcl application that the system is to use for the calls configured on the inbound dial peer. Enter the name to be associated with the Tcl IVR script.
- *script_url* is the pathname where the script is stored. Enter the pathname of the storage location first and then the script filename. Tcl IVR scripts can be stored in Flash memory or on a server that is acceptable using a URL, such as a TFTP server.
- *parameter value* allows you to configure values for specific parameters, such as language or PIN length.

For more information about the **call application voice** command, refer to *Interactive Voice Response Version 2.0 on Cisco VoIP Gateways*.

In the following example, the application named "test" is associated with the Tcl IVR script called newapp.tcl, which is located at tftp://keyer/debit_audio/:

```
(config)# call application voice test tftp://keyer/debit_audio/newapp.tcl
```

Note If the script cannot be loaded, it is placed in a retry queue and the system periodically retries to load it. If you modify your script, you can reload it using only the script name: **(config)# call application voice load** *script_name*

For more information about the **call application voice** and **call application voice load** commands, refer to *Interactive Voice Response Version 2.0 on Cisco VoIP Gateways*.

Associating Your Script with an Inbound Dial Peer

To invoke your Tcl IVR script to handle a call, you must associate the application configured with an inbound dial peer. To associate your script with an inbound dial peer, enter the following commands in configuration mode:

```
(config)# dial-peer voice number voip

(conf-dial-peer)# incoming called-number destination_number

(conf-dial-peer)# application application_name
```

In these commands:

- *number* uniquely identifies the dial peer. (This number has local significance only.)
- *destination_number* specifies the destination telephone number. Valid entries are any series of digits that specify the E.164 telephone number.
- *application_name* is the abbreviated name that you assigned when you loaded the application.

For example, the following commands indicate that the application called "newapp" should be invoked for calls that come in from an IP network and are destined for the telephone number of 125.

```
(config)# dial-peer voice 3 voip
(conf-dial-peer)# incoming called-number 125
(conf-dial-peer)# application newapp
```

For more information about inbound dial peers, refer to the Cisco IOS software documentation.

Displaying Information About IVR Scripts

To view a list of the voice applications that are configured on the router, use the **show call application voice** command. A one-line summary of each application is displayed.

```
show call application voice [ [name] | [summary] ]
```

In this command:

- *name* indicates the name of the desired IVR application. If you enter the name of a specific application, the system supplies information about that application.
- **summary** indicates that you want to view summary information. If you specify the summary keyword, a one-line summary is displayed about each application. If you omit this keyword, a detailed description of the specified application is displayed.

The following is an example of the output of the **show call application voice** command:

```
router# show call application voice session2
Idle call list has 0 calls on it.
Application session2
    The script is read from URL tftp://dirt/sarvi/scripts/tcl/app_session.tcl
    The uid-len is 10               (Default)
    The pin-len is 4                (Default)
    The warning-time is 60          (Default)
    The retry-count is 3            (Default)
    It has 0 calls active.

The Tcl Script is:
------------------
# app_session.tcl
#-----------------------------------------------------------------
```

```
# August 1999, Saravanan Shanmugham
#
# Copyright (c) 1998, 1999 by cisco Systems, Inc.
# All rights reserved.
#------------------------------------------------------------------
#
# This tcl script mimics the default SESSION app
#
#
# If DID is configured, just place the call to the dnis
# Otherwise, output dial-tone and collect digits from the
# caller against the dial-plan.
#
# Then place the call. If successful, connect it up, otherwise
# the caller should hear a busy or congested signal.

# The main routine just establishes the state machine and then exits.
# From then on the system drives the state machine depending on the
# events it receives and calls the appropriate tcl procedure


#---------------------------------
#   Example Script
#---------------------------------


proc init { } {
    global param

    set param(interruptPrompt) true
    set param(abortKey) *
    set param(terminationKey) #

}

proc act_Setup { } {
    global dest
    global beep

    set beep 0
    leg setupack leg_incoming

    if { [infotag get leg_isdid] } {
        set dest [infotag get leg_dnis]
        leg proceeding leg_incoming
        leg setup $dest callInfo leg_incoming
        fsm setstate PLACECALL
    } else {

        playtone leg_incoming tn_dial

        set param(dialPlan) true
        leg collectdigits leg_incoming param
      }


}

proc act_GotDest { } {
    global dest

    set status [infotag get evt_status]

    if {  $status == "cd_004" } {
```

```
        set dest [infotag get evt_dcdigits]
        leg proceeding leg_incoming
        leg setup $dest callInfo leg_incoming

    } else {
        puts "\nCall [infotag get con_all] got event $status while placing an outgoing
call"
        call close
    }
}


proc act_CallSetupDone { } {
    global beep

    set status [infotag get evt_status]

    if { $status == "CS_000"} {

        set  creditTimeLeft [infotag get leg_settlement_time leg_outgoing]

        if { ($creditTimeLeft == "unlimited") ||
             ($creditTimeLeft == "uninitialized") } {
            puts "\n Unlimited Time"
        } else {
            # start the timer for ...
            if { $creditTimeLeft < 10 } {
                set beep 1
                set delay $creditTimeLeft
            } else {
                set delay [expr $creditTimeLeft - 10]
            }
            timer start leg_timer $delay leg_incoming
        }
    } else {
        puts "Call [infotag get con_all] got event $status collecting destination"
        call close
    }
}

proc act_Timer { } {
    global beep
    global incoming
    global outgoing

    set incoming [infotag get leg_incoming]
    set outgoing [infotag get leg_outgoing]

    if { $beep == 0 } {
        #insert a beep ...to the caller
        connection destroy con_all
        set beep 1
    } else {
        media play leg_incoming flash:out_of_time.au
        fsm setstate CALLDISCONNECTED
    }
}

proc act_Destroy { } {
    media play leg_incoming flash:beep.au
}

proc act_Beeped { } {
    global incoming
```

```
    global outgoing

    connection create $incoming $outgoing
}

proc act_ConnectedAgain { } {
    timer start leg_timer 10 leg_incoming
}

proc act_Ignore { } {
# Dummy
    puts "Event Capture"
}

proc act_Cleanup { } {
    call close
}

init

#---------------------------------
#   State Machine
#---------------------------------
  set TopFSM(any_state,ev_disconnected) "act_Cleanup,same_state"
  set TopFSM(CALL_INIT,ev_setup_indication) "act_Setup,GETDEST"
  set TopFSM(GETDEST,ev_digitcollect_done) "act_GotDest,PLACECALL"
  set TopFSM(PLACECALL,ev_setup_done)  "act_CallSetupDone,CALLACTIVE"
  set TopFSM(CALLACTIVE,ev_leg_timer) "act_Timer,INSERTBEEP"
  set TopFSM(INSERTBEEP,ev_destroy_done) "act_Destroy,same_state"
  set TopFSM(INSERTBEEP,ev_media_done)  "act_Beeped,same_state"
  set TopFSM(INSERTBEEP,ev_create_done)    "act_ConnectedAgain,CALLACTIVE"
  set TopFSM(CALLACTIVE,ev_disconnected)   "act_Cleanup,CALLDISCONNECTED"
  set TopFSM(CALLDISCONNECTED,ev_disconnected) "act_Cleanup,same_state"
  set TopFSM(CALLDISCONNECTED,ev_media_done)  "act_Cleanup,same_state"
  set TopFSM(CALLDISCONNECTED,ev_media_done)  "act_Cleanup,same_state"
  set TopFSM(CALLDISCONNECTED,ev_disconnect_done) "act_Cleanup,same_state"
  set TopFSM(CALLDISCONNECTED,ev_leg_timer) "act_Cleanup,same_state"


  fsm define TopFSM  CALL_INIT
```

Using URLs in IVR Scripts

With IVR scripts, you use URLs to call the script and to call the audio files that the script plays. The VoIP system uses Cisco IOS File System (IFS) to read the files, so any IFS supported URLs can be used, which includes TFTP, FTP, or a pointer to a device on the router.

Note There is a limit of 32 entries in Flash memory, so you may not be able to copy all your audio files into Flash memory.

URLs for Loading the IVR Script

The URL of the IVR script is a standard URL that points to the location of the script. Examples include:

- flash:myscript.tcl—The script called myscript.tcl is being loaded from Flash memory on the router.
- slot0:myscript.tcl—The script called myscript.tcl is being loaded from a device in slot 0 on the router.
- tftp://BigServer/myscripts/betterMouseTrap.tcl—The script called myscript.tcl is being loaded from a server called BigServer in a directory within the tftpboot directory called myscripts.

URLs for Loading Audio Files

URLs for audio files are different from those used to load IVR scripts. With URLs for audio files:

- For static prompts, you can use the IFS-supported URLs as described in the "URLs for Loading the IVR Script" section on page 2-14.
- For dynamic prompts, the URL is created by the software, using information from the parameters specified for the **media play** command and the language CLI configuration command.

Tips for Using Your Tcl IVR Script

This section provides some answers to frequently asked questions about using Tcl IVR scripts.

- How do I get information from my RADIUS server to the Tcl IVR script?

  After you have performed an authentication and authorization, you can use the **infotag get** command to obtain the credit amount, credit time, and cause codes maintained by the RADIUS server.

- What happens if my script encounters an error?

  When an error is encountered in the script, the call is cleared with a cause of TEMPORARY_FAILURE (41). If the IVR application has already accepted the incoming call, the caller hears silence. If the script has not accepted the incoming call, the caller might hear a fast busy signal.

  If the script exits with an error and IVR debugging is on (as described in the "Testing and Debugging Your Script" section on page 2-8), the location of the error in the script is displayed at the command line.

Tcl IVR API Command Reference

This chapter provides an alphabetical listing of the Tcl IVR API commands and includes the following topics:

The following is provided for each command:

- Description of the purpose or function of the command
- Description of the syntax
- List of arguments and a description of each
- List of the possible return values and a description of each
- List of events received upon command completion
- Example of how the command can be used

For information about returns and events, see Chapter 5, "Events and Status Codes."

Standard Tcl Commands Used in Tcl IVR Scripts

The following standard Tcl 8.3.4 commands can be used in Tcl IVR 2.0 scripts:

Table 3-1 Standard Tcl Commands supported by Cisco IVR 2.0

append	array	binary	break
case	catch	cd	clock
close	concat	continue	encoding
eof[1]	error	eval	expr
fconfigure[1]	file[2]	fileevent	flush
for	foreach	format	gets[1]
glob	global	history	if
incr	info	join	lappend
lindex	linsert	list	llength
lrange	lreplace	lsearch	lsort
namespace	open	package[3]	proc

Table 3-1 Standard Tcl Commands supported by Cisco IVR 2.0

puts	pwd	read	regexp
regsub	rename	return	scan
seek	set	split	string
subst	switch	tcl_trace	time
unset	update	uplevel	upvar
variable	while	– –	– –

1. *ChannelId* must be an identifier for an open channel, such as a Tcl standard channel (stdin, stdout, or stderr) or the return value from an invocation of open.
2. The **file** *readlink* option is not supported. For **file** *attributes*, only group, owner, and permissions are supported.
3. The **package** *ifneeded* and **package** *unknown* options are not supported.

Note For the **puts** command, the display is limited to a character size of 2K.

HTTP Commands

The **http** package is included with Tcl 8.3.4 and provides the client side of the HTTP1.0 protocol. Table 3-2 identifies HTTP commands that are used in Tcl IVR 2.0 scripts with commands and options that are not supported on Cisco IOS .

Table 3-2 HTTP Commands

Supported Commands	Unsupported Commands	Unsupported Options
config		*-proxyhost* hostname
		-proxyport number
		-proxyfilter command
geturl		*-channel* name
		-handler callback
		-blocksize size
		-progress callback
		-queryblocksize size
		-queryprogress callback
formatQuery		
reset		
	wait	
status		
size		
code		
ncode		
data		

Table 3-2 HTTP Commands

Supported Commands	Unsupported Commands	Unsupported Options
error		
cleanup		
	register	
	unregister	

Tcl IVR Commands At a Glance

In addition to the standard Tcl commands, you can use the Tcl IVR 2.0 extensions created by Cisco. Also, Cisco modified the existing **puts** Tcl command to perform specific tasks. Table 3-2.

Table 3-3 Tcl IVR Commands

Command	Description
aaa accounting	Sends start or update accounting records
aaa accounting get status	Queries the accounting status of the leg or retrieves the status of any method list.
aaa accounting probe	Sends an accounting probe record.
aaa accounting set status	Changes the method list status.
aaa authenticate	Sends an authentication request to an external system, typically a Remote Access Dial-In User Services (RADIUS) server.
aaa authorize	Sends an authorization request to an external system, typically a RADIUS server.
call close	Marks the end of the call, releases all resources associated with that call, and frees the execution instance to handle the next call.
call lookup	Retrieves the application handle of an application that has registered for calls matching the specified parameters.
call register	Used by an application to indicate that it wants to receive any future incoming calls that match the specified call criteria. It also enables another application to lookup and retrieve this application's instance handle by matching the call criteria.
call unregister	Removes the call-registration entries for the specified call criteria.
clock	Performs one of several operations that can obtain or manipulate strings or values that represent some amount of time.
command terminate	Terminates a previously issued command.
connection create	Connects two call legs.
connection destroy	Destroys a connection.
command export	Allows the Tcl script to register or export a Tcl procedure to be invoked from C-code through a dynamic linking mechanism.
fsm define	Registers a state machine specified by a Tcl array and its start state.

Table 3-3 Tcl IVR Commands (continued)

Command	Description
fsm setstate	Specifies the next state of the FSM after completion of the current action procedure.
handoff	Hands off the name or handle of the application.
handoff return	Returns the call leg to the application.
infotag get	Retrieves information from a call leg, script, or system.
infotag set	Allows you to set information in the system.
leg alert	Sends an alert message to the specified leg.
leg callerid	Sends an updated call number and name after a transfer.
leg collectdigits	Moves the call into Digit Collect mode and collects the digits.
leg connect	Sends a call connect message to the incoming call leg.
leg consult abandon	Sends a call transfer consultation abandon request on the specified leg.
leg consult response	Sends a call transfer consultation identifier response on the specified leg.
leg consult request	Sends a call transfer consultation identifier request on the specified leg.
leg disconnect	Disconnects one or more call legs that are not part of a connection.
leg disconnect_prog_ind	Sends a disconnect message with the specified progress indicator value to the specified leg.
leg facility	Originates a facility message.
leg proceeding	Sends a call proceeding message to the incoming call leg.
leg progress	Sends a progress message to the specified leg.
leg senddigit	Transmits a digit on the specified call leg.
leg sendhookflash	Transmits a hook flash on the specified call leg.
leg setup	Initiates an outgoing call setup to the destination number.
leg setup_continue	Initiate a setup to an endpoint address or lets the system continue its action after an event interrupts the call processing.
leg setupack	Sends a call setup acknowledgement back to the incoming call leg.
leg tonedetect	Enables or disables the detection of specific tones during a call.
leg transferdone	Indicates the status of the call transfer on a call-leg and disconnects the call-leg.
leg vxmldialog	Initiates a VoiceXML dialog on the specified leg.
leg vxmlsend	Throws an event at an ongoing VoiceXML dialog on the leg.
log	Originates a syslog message.
media pause	Pauses the prompt playing on a specific call leg.
media play	Plays a prompt on a specific call leg.
media record	Records the the audio received on the specified call leg and saves it to the location specified by the URL.
media resume	Resumes play of a prompt on a specific call leg.

Table 3-3 Tcl IVR Commands (continued)

Command	Description
media seek	Seeks forward or backward in the current prompt.
media stop	Stops the prompt playing on a specific call leg.
modulespace	Allows the creation, access, and deletion of a modulespace in which a module can execute code.
object create dial-peer	Creates a list of dial-peer handles.
object create gtd	Creates a GTD Handle to a new GTD area from scratch.
object destroy	Destroys one or more dial peer items.
object append gtd	Appends one or more GTD attributes to a handle.
object delete gtd	Deletes one or more GTD attributes.
object replace gtd	Replaces one or more GTD attributes.
object get gtd	Retrieves the value of an attribute instance or a list of attributes associated with the specified GTD handle.
object get dial-peer	Returns dial peer information of a dial peer item or a set of dial peers.
param read	Reads configuration parameters associated with the call into a variable with the name *<variable-name>*, which becomes read-only.
param register	Registers a parameter, with description and default values, allowing them to be configured and validated through the CLI.
phone assign	Plays a specific tone or one according to the status code provided on a call leg.
phone query	Plays a specific tone or one according to the status code provided on a call leg.
phone unassign	Plays a specific tone or one according to the status code provided on a call leg.
playtone	Plays a specific tone or one according to the status code provided on a call leg.
puts	Prints the parameter to the console. Used for debugging.
requiredversion	Verifies the current version of the Tcl IVR API.
sendmsg	Sends a message to another application instance.
service	Registers or unregisters a service.
set avsend	Sets an associative array containing standard AV or VSA pairs.
set callinfo	Sets the parameters in an array that determines how the call is placed.
subscription open	Sends a subscription request to a subscription server.
subscription close	Removes an existing subscription.
subscription notify_ack	Sends a positive or negative acknowledgment for a notification event.
timer left	Returns the time left on an active timer.
timer start	Starts a timer for a call on a specific call leg.
timer stop	Stops the timer.

Tcl IVR Commands

The following is an alphabetical list of available Tcl IVR commands.

aaa accounting

The **aaa accounting** command sends start or update accounting records.

Note There is no stop verb. The stop record should always be generated automatically because of data availability. Use the update verb to add additional AVs to the stop record.

Syntax

aaa accounting start {*legID* | *info-tag*} [-a *avlistSend*][-s *servertag*][-t *acctTempName*]

aaa accounting update {*legID* | *info-tag*} [-a *avlistSend*]

Arguments

- *legID*—The call leg id (incoming or outgoing).
- *info-tag*—A direct mapped info-tag mapping to one leg. For more information on information tags, see Chapter 4, "Information Tags."
- -s *servertag*—The server (or server group)'s identifier. This value refer to the *method-list-name* as in AAA configuration:

  aaa accounting connection {**default** | *method-list-name*} **group** *group-name*

  Default value is `h323` (backward-compatible).
- -t *acctTempName*—Choose an accounting template which defines what attributes to send to the RADIUS server.
- -a *avlistSend*—Specify a list of av-pairs to append to the accounting buffer, which will be sent in the accounting record, or replace existing one(s) if the attribute in the list has a **r** flag associated with it. For example:

  ```
  set avlistSend(h323-credit-amount, r) 50.
  ```

Return Values

None.

Command Completion

Immediate.

Examples

```
aaa accounting start leg_incoming -a avList -s $method -t $template
aaa accounting update leg_incoming -a avList
```

Usage Notes

- After a start packet is issued, a corresponding stop packet is issued regardless of any suppressing configuration.

- If **debug voip aaa** is enabled and an accounting start packet has already been issued, either by the VoIP infrastrucure (enabled by Cisco IOS configuration command **gw-accounting aaa**) or execution of this Tcl verb in the script, the start request is ignored and a warning message is issued.
- If **debug voip aaa** is enabled and the **update** verb is called before start, the request is ignored and a warning message is issued.
- Although the original intent of this option is for additional application-level attributes (which are only known by the script rather than the underlying VoIP infrastructure) in the accounting packet, all the AAA attributes that can be included in an accounting request can be sent by using the **-a** option. Only the following list of attributes are supported for use in this manner with the **-a** option, although there is no sanity checking:
  - h323-ivr-out
  - h323-ivr-in
  - h323-credit-amount
  - h323-credit-time
  - h323-return-code
  - h323-prompt-id
  - h323-time-and-day
  - h323-redirect-number
  - h323-preferred-lang
  - h323-redirect-ip-addr
  - h323-billing-model
  - h323-currency

  There is also no sanity check if an attribute is only allowed to be included once. It is the responsibility of the script writer to maintain such integrity.

aaa accounting get status

The **aaa accounting get status** command queries the accounting status of the leg or retrieves the status of any method list.

Syntax

aaa accounting get status *{***-l** *<legID | info-tag>* | **-m** *method-list-name}*

Arguments

- **-l** *legID*—The call leg ID.
- **-l** *info-tag*—A direct-mapped information tag that maps to one leg.
- **-m** *method-list-name*—The server or server group identifier. This value refers to the method-list-name, as in the following AAA configuration:

```
aaa accounting connection {default | method-list-name} group group-name
```

Return Values

This command returns the following:

- unreachable—The accounting status is unreachable.

- reachable—The accounting status is reachable.
- unknown—The accounting status is unknown. If the monitoring of RADIUS-server connectivity is not enabled, the default status is unknown.
- invalid—The method list or legID specified is invalid.

Command Completion

Immediate

Examples

```
aaa accounting get status -l leg_incoming
aaa accounting get status -l [infotag get evt_leg]
set m1_1_status [aaa accounting get status -m m1_1]
```

Usage Notes

- This command only takes one leg, not multiple legs.
- The **-l** and **-m** options are mutually exclusive. If one is specified, the other should not be.

aaa accounting probe

The **aaa accounting probe** command sends an accounting probe record.

Syntax

aaa accounting probe <-s *servertag***>** ***[-a*** *avlistSend] [-t recordType]*

Arguments

- **-s** *servertag*—The server or server group identifier. This value refers to the method-list name, as in the following AAA configuration:

```
aaa accounting connection {default | method-list-name} group group-name
```

- **-a** *avlistSend*—Specifies a list of av-pairs to append to the accounting buffer to be sent in the accounting record.
- **-t** *recordType*—Specifies a *start*, *stop*, or *accounting-on* accounting record type.

Return Values

probe success—Probing is successful.

probe failed—Probing failed.

Command Completion

Immediate

Examples

```
aaa accounting probe -s m1_1

set av_send(username) "1234567890"
aaa accounting probe -s m1_1 -a av_send -t stop
```

Usage Notes

This command sends a dummy accounting probe record.

aaa accounting set status

The **aaa accounting set status** command changes the method list status.

Syntax

aaa accounting set status *method-list-status method-list-name*

Arguments

- *method-list-status*—Sets the server status. Possible values are:
  - unreachable—The server status is unreachable.
  - reachable—The server status is reachable.
- *method-list-name*—The server or server group identifier. This value refers to the method-list-name, as in the following AAA configuration:

```
aaa accounting connection {default | method-list-name} group group-name
```

Return Values

invalid—The method list specified is invalid.

unknown—The method list specified is not subscribed for status monitoring.

reachable—The method list specified is successfully set to the reachable state.

unreachable—The method list specified is successfully set to the unreachable state.

Command Completion

Immediate

Examples

```
set m1_status "unreachable"
aaa accounting set status reachable m1_1
aaa accounting set status unreachable m1_2
```

Usage Notes

This command sets the status of the specified method list.

aaa authenticate

The **aaa authenticate** command validates the authenticity of the user by sending the account number and password to the appropriate server for authentication. This command returns an accept or reject; it does not support the **infotag get aaa-avpair** *avpair-name* command for retrieving information returned by the RADIUS server in the authentication response.

Syntax

aaa authenticate *account password* [-a *avlistSend*][-s *servertag*][-l *legID*]

Arguments

- *account*—The user's account number.
- *password*—The user's password (or PIN).

- -a *avlistSend*—This argument is a replacement for the existing [*av-send*] optional argument. Backward-compatibility is provided.
- -s servertag—The server (or server group)'s identifier. This value refers to the *method-list-name* as in AAA configuration:

  aaa authentication login {**default** | *method-list-name*} **group** *group-name*

  Default value is `h323` (backward-compatible).

  Note Only general-purpose AAA server is currently supported.

- -l legID—The call leg for the access request. Causes voice-specific attributes (VSAs) associated with the call leg, such as h323-conf-id, to be packed into the access request.

Return Values

None

Command Completion

When the command has finished, the script receives an ev_authenticate_done event.

Example

```
aaa authenticate $account $password -a $avlistSend -s $method -l leg_incoming
```

Usage Notes

- Typically a RADIUS server is used for authentication, but any AAA-supported method can be used.
- If Tcl IVR command debugging is on (see the "Testing and Debugging Your Script" section on page 2-8), the account number and password are displayed.
- Account numbers and PINs are truncated to 32 characters, the E.164 maximum length.
- You can use the **aaa authentication login** and **radius-server** commands to configure a number of RADIUS parameters. For more information, see "Authentication, Authorization, and Accounting (AAA)", *Cisco IOS Security Configuration Guide*, Release 12.2, located at http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122cgcr/fsecur_c/index.htm
- To define avSend, see set avsend, page 3-72.
- If the -l option is not specified, the h323-conf-id attribute may not be included in the access request.

aaa authorize

The **aaa authorize** command sends a RADIUS authentication or authorization request, and allows the Tcl IVR script to retrieve information that the RADIUS server includes in its response. The command can be used multiple times during a single call (for example, to do the authentication, then to do the authorization).

When used in combination with the **aaa authenticate** command, this command provides additional information to the RADIUS server, such as the destination and origination numbers, after a user has been successfully authenticated. When used both to authenticate and authorize the user, the values used in the command's parameters are altered to support each intended purpose. Parameters can be left blank (null), as illustrated in the examples.

Syntax

aaa authorize *account password ani destination* {*legID* | *info-tag*} [-a *avlistSend*] [-s servertag] [-g *GUID*]

Arguments

- *account*—User's account number.
- *password*—User's password (or PIN).
- *ani*—Origination (calling) number.
- *destination*—Call destination (called) number.
- *legID*—ID of the incoming call leg.
- *info-tag*—A direct mapped info-tag mapping to one leg. For more information about info-tags, see Chapter 4, "Information Tags."
- -a *avlistSend*—This argument is a replacement for the existing [*av-send*] optional argument. Backward-compatibility is provided.
- -s *servertag*—The server (or server group) identifier. This value refers to the *method-list-name* as in AAA configuration:

  aaa authentication exec {**default** | *method-list-name*} **group** *group-name*

  Default value is *h323* (backward-compatible).
- -g *GUID*—Specifies the GUID to use in the authorize operation.

The *account* and *password* arguments are the same as those specified in the **aaa authenticate** command. The *destination* and *ani* arguments provide additional information to the external server.

Return Values

None

Command Completion

When the command finishes, the script receives an ev_authorize_done event.

Examples

```
aaa authorize $account $password $ani $destination $legid
aaa authorize $account "" $ani "" $legid
aaa authorize $ani "" $ani "" $legid
aaa authorize $account $pin $ani $destination $legid  -a avList -s $method -t $template
```

Usage Notes

- Additional parameters can be returned by the RADIUS server as attribute-value (AV) pairs. To determine whether additional parameters have been returned, use the aaa_avpair_exists info-tag. Then to read the parameters, use the aaa_avpair info-tag. For more information about info-tags, see Chapter 4, "Information Tags."
- If Tcl IVR commands debugging is on (see the "Testing and Debugging Your Script" section on page 2-8), the account number, password, and destination are displayed.
- Account numbers, PINs, and destination numbers are truncated at 32 characters, the E.164 maximum length.
- If the specified call leg is invalid, the script terminates and displays an error to the console, and the call is cleared.

- You can use the **aaa authentication login** and **radius-server** commands to configure a number of RADIUS parameters. For more information, see "Authentication, Authorization, and Accounting (AAA)," *Cisco IOS Security Configuration Guide,* Release 12.2, located at http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122cgcr/fsecur_c/index.htm
- To define avSend, see set avsend, page 3-72.

call close

The **call close** command marks the end of the call and frees the execution instance of the script to handle the next call. This command causes the system to clean up the resources associated with that call. If conferenced legs exist, this command destroys the connections and clears all the call legs. If **leg collectdigits** is active on any of the call legs, the digit collection process is terminated and the call is cleared.

Syntax

call close [-r]

Arguments

-r—Retains the subscriptions pertaining to the application.

Return Values

If a call is closed using the -r argument, the resources used by that instance are freed, but any subscriptions created with the **subscribe** command remain active and allow notifications to start a session. If a notification comes in for a retained subscription after an instance closes with a -r argument, a new instance can be generated to handle the notification.

Note A new session starts only if the original subscription request, sent with the **subscription open** command, specifies a configured application as *notificationReceiver*.

Command Completion

Immediate

Examples

```
proc act_Disconnected {} {
call close -r
}
set FSM(any_state,ev_disconnected) "act_Disconnected, CALL_CLOSED"

proc act_UnsubscribeDone {} {
call close
}
set fsm(any_state,ev_unsubscribe_done) "act_UnsubscribeDone SUBS_OVER"
```

Usage Notes

- The **call close** command marks the end of the call and the end of the script. This command causes the system to clean up the resources. If the **call close** command is called without the -r option, the subscription is removed from the server before closing the running instance.
- When using the **call close -r** command, make sure *notificationReceiver*, a configured application or application handle, is specified.

call lookup

The **call lookup** command retrieves the application handle of an application that has registered for calls matching the specified parameters.

Syntax

call lookup *matchParam*

Arguments

- *matchParam*—An associative array containing the call parameters that describe the calls this application is registering for. Supported call parameters are:
  - calledNum—Value of the called number of an incoming call to be matched.
  - transferConsultID—Value of the call transfer consultation identifier of an incoming call to be matched.

Return Values

Returns an application handle if another application has registered for calls matching the specified parameters. Returns a null string if no application has registered for calls matching the specified parameters.

Command Completion

Immediate

Examples

```
set matchParam(calledNum) $calledDNIS
set matchParam(transferConsultID) $consultID
set handler [call lookup matchParam]
```

Usage Notes

A call registration entry must match all specified *matchParam* parameters to be considered a successful match. If the application does not specify any *matchParam* parameters, the script terminates, an error is printed to the console, and the call is cleared.

call register

The **call register** command is used by an application to indicate that it wants to receive any future incoming calls that match the specified call criteria. It also enables another application to lookup and retrieve this application's instance handle by matching the call criteria. See the **call lookup** command for more information.

Syntax

call register *matchParam* [-i]

Arguments

- *matchParam*—An associative array containing the call parameters that describe the calls this application is registering for. Supported call parameters are:
  - calledNum—Value of the called number of an incoming call to be matched.
  - transferConsultID—Value of the call transfer consultation identifier of an incoming call to be matched.

- -i—Disable automatic call routing. If specified, the application does not receive the incoming call even if the specified call parameters match. This is useful when an application wants other applications to hand off call matching of the specified call parameters.

Return Values

0—Registration success

1—Registration failed, duplicate entry

Command Completion

Immediate

Examples

```
set matchParam(calledNum) $calledDNIS
set matchParam(transferConsultID) $consultID
set registerStatus [call register matchParam -i]
```

Usage Notes

- This command fails if another application has already registered for calls matching the same call parameters.
- When an application successfully invokes the **call register** command, any future incoming calls that match all parameters specified in the *matchParam* parameter results in a match.
- By default, a matched incoming call is immediately routed to the registered application and this application receives an ev_setup_ind event.
- If the call registration command specifies the -i parameter, no calls automatically route to this application. Instead, the application should be prepared to receive an *ev_handoff* event from another application. See the **call lookup** command usage notes for more information.
- If the application specifies an invalid argument, the script terminates, an error is printed to the console, and the call is cleared.
- If the application does not specify any *matchParam* parameters, the script terminates, an error is printed to the console, and the call is cleared.

call unregister

The **call unregister** command removes the call-registration entries for the specified call criteria.

Syntax

call unregister *matchParam*

Arguments

- *matchParam*—An associative array containing the call parameters that describe the calls this application is registering for. Supported call parameters are:
  - calledNum—Value of the called number of an incoming call to be matched.
  - transferConsultID—Value of the call transfer consultation identifier of an incoming call to be matched.

Return Values

0—Unregistration success

1—Unregistration failed, entry not available

Command Completion

Immediate

Examples

```
set matchParam(calledNum) $calledDNIS
set matchParam(transferConsultID) $consultID
set unregisterStatus [call unregister matchParam]
```

Usage Notes

- This command is used by an application when it no longer wants to receive calls that it previously registered for. A call registration entry must match all specified *matchParam* parameters to be unregistered by this command.
- If the application does not specify any matchParam parameters, the script terminates, an error is printed to the console, and the call is cleared.

clock

This command performs one of several operations that can obtain or manipulate strings or values that represent some amount of time.

Syntax

clock *option arg arg*

Arguments

- *option*—Valid options are:
  - **clicks**—Return a high-resolution time value as a system-dependent integer value. The unit of the value is system-dependent, but should be the highest resolution clock available on the system, such as a CPU cycle counter. This value should only be used for the relative measurement of elapsed time.
  - **format** *clockValue -format string -gmt boolean*—Converts an integer time value, typically returned by **clock** *seconds*, **clock** *scan*, or the *atime*, *mtime*, or *ctime* options of the **file** command, to human-readable form. If the -format argument is present the next argument is a string that describes how the date and time are to be formatted. Field descriptors consist of a % followed by a field descriptor character. All other characters are copied into the result. Valid field descriptors are:
  - %%—Insert a %.
  - %a—Abbreviated weekday name (Mon, Tue, etc.).
  - %A—Full weekday name (Monday, Tuesday, etc.).
  - %b—Abbreviated month name (Jan, Feb, etc.).
  - %B—Full month name.
  - %c—Locale specific date and time.
  - %d—Day of month (01 - 31).

- %H—Hour in 24-hour format (00 - 23).
- %I—Hour in 12-hour format (00 - 12).
- %j—Day of year (001 - 366).
- %m—Month number (01 - 12).
- %M—Minute (00 - 59).
- %p—AM/PM indicator.
- %S—Seconds (00 - 59).
- %U—Week of year (01 - 52), Sunday is the first day of the week.
- %w—Weekday number (Sunday = 0).
- %W—Week of year (01 - 52), Monday is the first day of the week.
- %x—Locale specific date format.
- %X—Locale specific time format.
- %y—Year without century (00 - 99).
- %Y—Year with century (for example, 2002)
- %Z—Time zone name.

In addition, the following field descriptors may be supported on some systems. For example, UNIX but not Microsoft Windows. Cisco IOS software supports the following options:

- %D—Date as %m/%d/%y.
- %e—Day of month (1 - 31), no leading zeros.
- %h—Abbreviated month name.
- %n—Insert a newline.
- %r—Time as %I:%M:%S %p.
- %R—Time as %H:%M.
- %t—Insert a tab.
- %T—Time as %H:%M:%S.

If the *-format* argument is not specified, the format string "%a %b %d %H:%M:%S %Z %Y" is used. If the *-gmt* argument is present, the next argument must be a boolean, which if true specifies that the time will be formatted as Greenwich Mean Time. If false then the local time zone will be used as defined by the operating environment.

- **scan** *dateString -base clockVal -gmt boolean*—Converts *dateString* to an integer clock value (see **clock seconds**). The **clock scan** command parses and converts virtually any standard date and/or time string, which can include standard time zone mnemonics. If only a time is specified, the current date is assumed. If the string does not contain a time zone mnemonic, the local time zone is assumed, unless the *-gmt* argument is true, in which case the clock value is calculated relative to Greenwich Mean Time.

  If the *-base* flag is specified, the next argument should contain an integer clock value. Only the date in this value is used, not the time. This is useful for determining the time on a specific day or doing other date-relative conversions.

  The *dateString* consists of zero or more specifications of the following form:

  - *time*—A time of day, which is of the form: *hh:mm:ss meridian zone* or *hhmm meridian zone*. If no meridian is specified, *hh* is interpreted on a 24-hour clock.

  - *date*—A specific month and day with optional year. The acceptable formats are mm/dd/yy, monthname dd, yy, dd monthname yy and day, dd monthname yy. The default year is the current year. If the year is less then 100, then 1900 is added to it.
  - *relative time*—A specification relative to the current time. The format is number units and acceptable units are *year*, *fortnight*, *month*, *week*, *day*, *hour*, *minute* (or *min*), and *second* (or *sec*). The unit can be specified in singular or plural form, as in *3 weeks*. These modifiers may also be specified: *tomorrow*, *yesterday*, *today*, *now*, *last*, *this*, *next*, *ago*.

  The actual date is calculated according to the following steps:

  - First, any absolute date and/or time is processed and converted. Using that time as the base, day-of-week specifications are added.
  - Next, relative specifications are used. If a date or day is specified, and no absolute or relative time is specified, midnight is used.
  - Finally, a correction is applied so that the correct hour of the day is produced after allowing for daylight savings time differences.

- **seconds**—Returns the current date and time as a system-dependent integer value. The unit of the value is seconds, allowing it to be used for relative time calculations. The value is usually defined as total elapsed time from an "epoch." The epoch should not be assumed.

Return Values

None

Command Completion

None

Example

```
set clock_seconds [clock seconds]
set time [clock format [clock seconds] -format "%H%M%S"]
set new_time [clock format [clock seconds] -format "%T"]
set time_hh [clock format [clock seconds] -format "%H"]
set date [clock format [clock seconds] -format "%Y%m%d"]
set new_date [clock format [clock seconds] -format "%D"]
set week [clock format [clock seconds] -format "%w"]
```

Usage Notes

None.

command export

The **command export** command lets the Tcl script register or export a Tcl procedure to be invoked from C-code through a dynamic linking mechanism.

Syntax

command export *<command-string> <command-template>*

Arguments

- *<command-string>*—The expanded name, including namespace information needed to invoke the procedure from outside its native namespace.

- *<command-template>*—A parameter template that the Tcl procedure accepts, allowing the DLL system to make sure the C-code that invokes this API calls it with the right type and number of parameters. This string is of the form *x:x:x:x:x:x*, where each *x* indicates the type of parameter permitted. The first *x* indicates the return type of the procedure. The value of *x* can be *s* to represent a string or char* parameter.

Return Values

None

Command Completion

Immediate

Examples

```
command export ::Service::handle_event s
```

Usage Notes

None

command terminate

The **command terminate** command ends or stops a previously issued command.

Syntax

command terminate *[commandHandle]*

Arguments

commandHandle—The handler handle associated with a handler retrieved by the **get last_command_handle** infotag. The leg setup command can be terminated using this verb. For more information about info-tags, see Chapter 4, "Information Tags."

Return Values

This command returns one of the following:

- *0 (pending)*—A command termination is initiated.
- *1 (terminated)*—The command termination has completed.
- *2 (failed)*—The command termination verb is not valid. Either the command argument is not correct, there is no such command pending, or the termination for that command has already been initiated.

Command Completion

If applied to a call setup verb, an ev_setup_done event is returned when the call setup handler terminates. The status code for this event is *ls_015*: terminated by application request.

Example

```
command terminate [$commandHandle]
```

Usage Notes

The last command handle has to be retrieved before any other command is issued.

connection create

The **connection create** command connects two call legs.

Syntax

connection create {*legID1* | *info-tag1*} {*legID2* | *info-tag2*}

Arguments

- *legID1*—The ID of the first call leg to be connected.
- *info-tag1*—A direct mapped info-tag mapping to one call leg. For more information about info-tags, see Chapter 4, "Information Tags."
- legID2—The ID of the second call leg to be connected.
- *info-tag2*—A direct mapped info-tag mapping to a single second leg. For more information about info-tags, see Chapter 4, "Information Tags."

Return Values

This command returns the following:

- *connectionID*—A unique ID assigned to this connection. This ID is required for the **connection destroy** command.

Command Completion

When this command finishes, the script receives an ev_create_done event.

Example

```
set connID [connection create $legID1 $legID2]
```

Usage Notes

- If the specified call leg is invalid, the script terminates and displays an error to the console, and the call is cleared.
- Connections between two IP legs are not supported. Even if the command seems to execute successfully, it actually does not work. Doing so could potentially cause problems, as there is currently no way to capture the resulting error at the script level. Therefore, it is advisable to avoid attempting such connections.
- If supplementary services such as hold or voice transfer are used, the called party can hear voice and media prompts being played from the calling party after the call legs have been destroyed. To avoid this problem, disable the **voice-fastpath enable** Cisco IOS command which is enabled by default. To disable it, use the **no voice-fastpath enable** global configuration command.

connection destroy

The **connection destroy** command destroys the connection between the two call legs.

Syntax

connection destroy {*connectionID* | *info-tag*}

Arguments

- *connectionID*—The unique ID assigned to this connection during the connection create process.

- *info-tag*—A direct mapped info-tag mapping to one connection ID. For more information about info-tags, see Chapter 4, "Information Tags."

Return Values

None

Command Completion

When this command finishes, the script receives an ev_destroy_done event.

Example

```
connection destroy $connID
```

Usage Notes

- The individual call legs are not disconnected; only the connection between the call legs is destroyed.
- If supplementary services such as hold or voice transfer are used, the called party can hear voice and media prompts being played from the calling party after the call legs have been destroyed. To work around this problem, disable the **voice-fastpath enable** Cisco IOS command which is enabled by default. To disable it, use the **no voice-fastpath enable** global configuration command.

fsm define

The **fsm define** command registers a state machine for the script. The state machine is specified using a Tcl array that lists the state event transition along with the appropriate action procedure.

Syntax

fsm define *statemachine_array start_state*

Arguments

- statemachine_array—An array that defines the state machine. The array is indexed by the current state and current event. The value of each entry is the action function to execute and the state to move to next. The format of the array entries is:

  set *statemachine_array*(*current_state*,*current_event*) "*actionFunction*,*next_state*"

  Note The current state and event are enclosed in parentheses and separated by a comma without any spaces. The resulting action and next state are enclosed in quotation marks and separated by a comma, spaces, or both.

- *start_state*—The starting state of the state machine. This is the state of script when a new call comes in for this script.

Return Values

None

Command Completion

Immediate

Example

```
#---------------------------------
#   State Machine
#---------------------------------
set FSM(CALL_INIT,ev_setup_indication) "act_Setup,DEST_COLLECT"

set FSM(DEST_COLLECT,ev_disconnect_done) "act_DCDone,CALL_SETTING"
set FSM(DEST_COLLECT,ev_disconnected) "act_DCDisc,CALL_DISCONNECTING"

set FSM(CALL_SETTING,ev_callsetup_done) "act_PCDone,CALL_ACTIVE"
set FSM(CALL_SETTING,ev_disconnected) "act_PCDisc,CALL_SETTING_WAIT"

fsm define FSM CALL_INIT
```

fsm setstate

The **fsm setstate** command allows you to specify the state to which the FSM moves to after completion of the action procedure.

Syntax

fsm setstate *StateName*

Arguments

- *StateName*—The state that the FSM should move to after the action procedure completes its execution. This overrides the next state specified in the current state transition of the FSM table.

Return Values

None

Command Completion

None

Example

```
#Check for DNIS, if there is DNIS you want to go to Call setup right away
set legID [infotag get evt_legs]
set destination [infotag get leg_dnis $legID]
if {destination != ""} {
          callProceeding $legID
          set callInfo(alertTime) 30
          call setup $destination callInfo leg_incoming
          #Moves to CALL_SETTING state
          fsm setstate CALL_SETTING
} else {
          leg setupack $legID
          playtone $legID TN_DIAL
          set DCInfo(dialPlan) true
          # Assumption: As per the state machine moves to DIGIT_COLLECT}
          leg collectdigits $legID DCInfo
}
```

Usage Notes

- This command allows the action procedure to specify the state that the FSM should move to (other than the state specified in the FSM table).

- If you do not use this command, the state transition follows the state machine as defined in the FSM table.

handoff

Hands off the name or handle of the application.

Syntax

handoff {*appl* | *callappl*} {*legID* | *info-tag*} [{*legID2* | *info-tag2*} ...] {*app-name* | <*handle*>} [**-s** <*argstring*>]

Arguments

- *appl* | *callappl*—Specific handoff command desired. The only difference is that *appl* does not provide a return value, *callappl* does.
- *legID* | *infotag*—The call leg ID to hand off to the destination. For more information about info-tags, see Chapter 4, "Information Tags."
- *app-name* | <*handle*>—The application name or handle.
- **-s** <*argstring*>—Information to pass to another application instance.

Return Values

If the handoff is to an instance that is not running, it returns an "unavailable" message.

Command Completion

Immediate.

Examples

```
set iid newapp
set sid customer
set anum 123456
handoff appl leg_incoming $iid -s "Here is a call: service=$sid; account number=$anum"

set iid newapp
set sid customer
set anum 123456
handoff callappl leg_incoming $iid -s "Here is a call: service=$sid; account number=$anum"
```

Usage Notes

- The application name is the name as configured in the **call app voice** <*name*> <*url*> configuration command and the application handle is the handle returned by either the *mod_handle_service* or the *evt_msg_source* information tags.
- This command can create a new instance by providing a name or it can try to hand off to an existing instance by providing a handle. A handle has a special internal format, which the system can parse to determine if it is a handle or a name. If the handle points to an instance that does not exist, the **handoff** command returns "unavailable". The script does not fail and still maintains control of the call legs.
- The application instance that receives the call leg can retrieve the argument string by using the evt_handoff_argstring information tag.
- If the handle points to an instance that does not exist, the handoff command returns "unavailable". The script does not fail and still maintains control over the call legs.

handoff return

Returns a set of separate call legs received from different sessions or a set of conferenced legs to the same session.

Syntax

handoff return *legID* [**-s** <*argstring*>]

Arguments

- *legID*—The call leg or legs to return. Can be a VAR_TAG, such as leg_incoming.
- **-s** <*argstring*>—Information to pass to another application instance.

Return Values

If the handoff is to an instance that is not running, it returns an "unavailable" message.

Command Completion

Immediate

Example

```
set leg2 leg_incoming
handoff return $leg2 -s "$sid; $anum"
```

Usage Notes

- The application instance that receives the call leg can retrieve the argument string by using the evt_handoff_argstring information tag.
- Handoff return of a set of separate call legs received from different sessions should be done with a separate **handoff return** command for each leg. The **handoff return** *leg_all* command is undefined in this case. The entire set of call legs returns to the return location for the first leg; however, which leg is listed first in the *leg_all* information tag is undefined.
- Handoff return of a set of conferenced legs returns both legs to the same session. For example, if a session has been handed leg1 from session1 and leg2 from session2, and it conferenced the two legs together. Then the command

  ```
  handoff return $leg2
  ```

  returns both legs, conferenced together, to session2.

infotag get

The **infotag get** command retrieves information from a call leg, call, script, or system. The information retrieved is based on the info-tag specified.

Syntax

infotag get *info-tag* [*parameter-list*]

Arguments

- *info-tag*—The info-tag that indicates the type of information to be retrieved. For more information about info-tags, see Chapter 4, "Information Tags."

- *parameter-list*—(Optional, depending on the info-tag) The list of parameters that further defines the information to be retrieved.

Return Values

The information requested.

Command Completion

Immediate

Example

```
set dnis [infotag get leg_dnis]
set language [infotag get med_language]
set leg_list2 [infotag get leg_legs]
```

Usage Notes

Some info-tags have specific scopes of access. For example, you cannot call evt_dcdigits while handling the ev_setup_done event. In other words, if the previous command is **leg setup** and the ev_setup_done event has not yet returned, then you cannot execute an **infotag get evt_dcdigits** command, or the script terminates with error output. For more information, see Chapter 4, "Information Tags.".

infotag set

The **infotag set** command allows you to set information in the system. This command works only with info-tags that are writable.

Syntax

infotag set {*info-tag* [*parameters*]} *value*

Arguments

- *info-tag*—The information to set. A list of info-tags that can be set is found in Chapter 4, "Information Tags," and are designated as "Write."
- *parameters*—A list of parameters that is dependent on the info-tag used.
- *value*—The value to set to. This is dependent on the info-tag used.

Return Values

None

Command Completion

Immediate

Example

```
infotag set med_language prefix ch
infotag set med_location ch 0 tftp://www.cisco.com/mediafiles/Chinese
```

leg alert

Sends an alert message to the specified leg.

Syntax

leg alert *{legID | info-tag}* [**-p** <*prog_ind_value*>] [**-s** <*sig_ind_value*>] [**-g** <*GTDHandle*>]

Arguments

- *legID | info-tag*—Points to the incoming leg to send the progress message to.
- **-s** <*sig_ind_value*>—The value of the call signal indication. The value is forwarded as is.
- **-p** <*prog_ind_value*>—The value of the call progress indication. The value is forwarded as is.
- **-g** <*GTD handle*>—The handle to a previously created GTD area. If not specified, the default is to send a ring back signal.

Return Values

None.

Command Completion

Immediate.

Examples

```
leg setupack leg_incoming
leg alert leg_incoming -s 1-g gtd_progress_handle
leg connect leg_incoming
```

Usage Notes

- Applications that terminate a call can insert a leg alert before connecting with the incoming leg to satisfy the switch.
- For the leg alert command to be successful, the leg must be in the proper state. The following conditions are checked on the target leg:
  - A leg setupack has been sent.
  - No leg alert has been sent.
- If the specified call leg is already in the connect state, the script terminates and displays an error to the console, and the call is cleared.

leg callerid

The **leg callerid** command allows an application to specify caller identification information to Cisco IP phones operating in a Cisco CallManager Express (CME) environment and analog FXS phones with the necessary caller ID capabilities.

Syntax

leg callerid *{legID | infotag} param*

Arguments

- *legID | infotag*—The call leg ID to hand off to the destination.
- *param*—An associative array containing the caller identification information for the specified call. There are two available parameters: *name* and *number. Number* is mandatory; *name* is optional.

Return Values

None.

Command Completion

Immediate.

Examples

```
set param(name) "Xee"
set param(number) "4088531936"
leg callerid param legXto

set param(name) "Xto"
set param(number) "4088531645"
leg callerid param legXee

set param(name) "John Smith"
set param(number) "1234567890"
leg callerid leg_outgoing param
```

Usage Notes

- If the specified call leg is invalid, the script terminates, an error is printed to the console, and the call is cleared.
- If param(number) is not specified, the script terminates, an error is printed to the console, and the call is cleared.
- If the **leg callerid** command is used for telephony call legs that do not have call waiting (with the exception of EFXS call legs), a beeping sound may be heard by the caller upon call connection. This beeping sound may confuse the caller because it usually indicates call waiting for analog FXS phones. To avoid confusion, use the **leg callerid** command only for EFXS call legs.
- Before using the **leg callerid** command, use the information tag **leg_type** to check the type of call leg.

leg collectdigits

The **leg collectdigits** command instructs the system to collect digits on a specified call leg against a dial plan, a list of patterns, or both.

Syntax

leg collectdigits {*legID* | *info-tag*} [*param* [*match*]]

Arguments

- *legID*—The ID of the call leg on which to enable digit collection.
- info-tag—A direct mapped info-tag mapping to one or more legs. For more information about info-tags, see Chapter 4, "Information Tags."
- param—An array of parameters that defines how the digits are to be collected. The array can contain the following:
  - param(**abortKey**)—Key to abort the digit collection. The default is none.
  - param(**interDigitTimeout**)—Interdigit timeout value in seconds. The default is 10.
  - param(**initialDigitTimeout**)—Initial digit timeout value in seconds. The default is 10.

- *param*(**interruptPrompt**)—Whether to interrupt the prompt when a key is pressed. Possible values are true and false. The default is false.
- *param*(**terminationKey**)—Key that terminates the digit collection. The default is none.
- *param* (**consumeDigit**)— Allows the application to prevent the digits dialed by the user from being relayed to a remote end point after the incoming and outgoing call legs are bridged.
- *param*(**dialPlan**)—Whether to match the digits collected against a dial plan (or pattern, if 1 is specified). Possible values are true and false. The default is false.
- *param*(**dialPlanTerm**)—Match incoming digits against a dial plan and, even if the match fails, continue to collect the digits until the termination key is pressed or a digit timeout occurs. Possible values are true and false. The default is false.
- *param*(**maxDigits**)—Maximum number of digits to collect before returning.
- *param*(**enableReporting**)—Whether to enable digit reporting when returning. Possible values are true and false. The default is false. After you have enabled digit reporting, the script receives an ev_digit_end event when each key is pressed and released. With digit reporting enabled, the script may also receive periodic *ev_digit_end* events with digit T, indicating an interdigit timeout, which usually can be ignored by the script.
- *param*(**ignoreInitialTermKey**)—This disallows or ignores the termination key as the first key in digit collection. The default is false.

- match—An array variable that contains the list of patterns that determines what the **leg collectdigits** command will look for. A %D string within a pattern string matches the corresponding digits against the dial plan.

Return Values

None

Command Completion

When the command finishes, the script receives an ev_collectdigits_done event, which contains the success or failure code and the digits collected. For more information about the success and failure codes, see the "Status Codes" section on page 5-6.

Examples

Example 1—Collect digits to match dialplan:

```
set params(interruptPrompt) true
set params(dialPlan) true
leg collectdigits $legID params
```

Example 2—Collect digits to match a pattern:

```
set pattern(1) "99.....9*"
set pattern(2) "88.....9*"
leg collectdigits $legID params pattern
```

Example 3—Collect digits to match a dial plan with a pattern prefix:

```
set pattern(1) "#43#%D"
leg collectdigits $legID params pattern
```

Example 4— Here is an example of using the **consumeDigit** parameter to prevent digit relay to a remote end point. The TCL application receives an *ev_digit_end* event for every dialed digit. None of these digits are relayed to the other call leg.

```
set param(enableReporting) true
```

```
set param(consumeDigit) true
leg collectdigits {legID|info-tag} param
```

Usage Notes

- If the specified call leg is invalid, the script terminates and displays an error to the console, and the call is cleared.
- By default, the script does not see any digits, because digit reporting is disabled on all call legs. For the script to see individual digit events, digit reporting must be turned on using the **leg collect digits** command with *parm*(**enableReporting**) set to TRUE.
- If enableReporting is set to TRUE, the command finishes and digit reporting remains on (allowing the script to receive the digits pressed). This is useful if you want the script to collect digits by itself or if you want to look for longpounds.
- If the **leg collectdigits** command is being issued just for enabling digit reporting, and is not expected to collect digits or patterns, the command will finish after it has turned reporting on. The script will receive the ev_collectdigits_done event with a status of cd_009.
- The initial timeout for collecting digits is 10 seconds and the interdigit collection timeout is 10 seconds. If the digit collection times out, a timeout status code along with the digits collected so far is returned. You can change the timeout values at the voice port using the **timeouts initial** and **timeout interdigit** commands.
- The **consumeDigit** parameter can be set to true or false (default).
  - Setting the **consumeDigit** parameter to true or false does not affect digit collection when the call leg is not bridged.
  - Setting the **consumeDigit** parameter to true does not prevent dialed digits from being passed to a remote end point if the negotiated DTMF relay is rtp-nte, cisco-rtp, or in-band voice.
- When multiple match criteria are specified for **leg collectdigits**, the matching preference order is maxDigits, dialPlan, pattern.

  The preference, maxDigits, is considered to be a special pattern.

  This special-pattern matching terminates and is considered to be a successful match if one of the following conditions occur:
  - The user dials the maximum number of digits.
  - The user presses the termination key, when set.
  - A time-out occurs after the user has dialed a few digits.

  When this happens, a cd_005 status code is reported. See Digit Collection Status, page 5-7.
- If the digits match the dialPlan with a pattern prefix, the command returns a pattern matched, cd_005, status code. See Digit Collection Status, page 5-7.
- %D dialPlan pattern matching string is allowed only at the end of the pattern. If %D is specified in any other position within the pattern, the script terminates, an error is sent to the console, and the call is cleared.
- If both a %D pattern is specified and the dialPlan parameter is set to TRUE, the command returns a dialplan matched, cd_004, status code on successful dialplan match. See Digit Collection Status, page 5-7.

  The *evt_dcpattern* and *evt_dcdigits* information tags can be used to retrieve the matched pattern and digits.

leg connect

The **leg connect** command sends a signaling level CONNECT message to the incoming call leg.

Syntax

leg connect {*legID* | *info-tag*}

Arguments

- *legID*—The ID of the incoming call leg to which the connect signaling message is sent.
- *info-tag*—A direct mapped info-tag mapping to one or more incoming legs. For more information about info-tags, see Chapter 4, "Information Tags."

Return Values

None

Command Completion

Immediate

Examples

```
leg connect leg_incoming
leg connect $legID
```

Usage Notes

- If the specified call leg is not incoming, the script terminates and displays an error to the console, and the call is cleared.
- If the info-tag specified maps to more than one incoming call leg, a call connect message is sent to all the incoming call legs that have not already received a call connect message.
- If the state of the specified call leg prevents it from receiving a call connect message (for example, if the state of the leg is disconnecting), the command fails.
- If the specified call leg is already in the connect state, the script terminates and displays an error to the console, and the call is cleared.

Note For incoming ISDN call legs, a setupack, proceeding, or alert message must be sent before the connect message. Otherwise, the script will receive an ev_disconnected event and the incoming leg will be disconnected.

leg consult abandon

This command is used to send a call-transfer consultation abandon request on the specified leg. Depending on the underlying protocol, the gateway may send a message to the endpoint. Typically, the endpoint cleans up its state and locally generates an error response indicating that the call transfer has failed.

Syntax

leg consult abandon *legID*

Arguments

legID—The ID of the call-leg to transfer-target endpoint.

Return Values

The command returns one of the following:

- *0 (success)*—The abandon message successfully sent on the call-leg
- *1 (failed, invalid state)*—The call-leg has not sent a consult request message earlier. It is invalid to send a consult-abandon message on a leg that has not sent a consult-request message.
- *2 (failed, protocol error)*—The abandon message could not be sent due to a protocol error.

Example

```
leg consult abandon $targetleg
set retcode [leg consult abandon $consultLeg]
```

Command Completion

Immediate

Related Events

None

leg consult response

This command is used to send a call-transfer consultation identifier response on the specified leg. A consult-id is automatically generated. Depending on the underlying protocol, the gateway either sends a message with the generated consult-id on the specified leg or ignores this command.

Syntax

leg consult response *legID {[-i consultID][-t transferDestNum]* | -c 'xxx'*}*

Arguments

- *legID*—ID of the call-leg to transferrer endpoint.
- *-i consultID*—consultation-id (optional)
- *-t transferDestNum*—transfer-target number. Diverted-to number could be used here when the transfer-target is locally forwarded to another number. If not specified, the legID's corresponding outgoing call leg's calledNumber is used. If an appropriate outgoing call leg does not exist, the legID's calledNumber is used. (optional)
- -c 'xxx'—Where 'xxx' is a consult failure code (optional)
  - 001—consultation failure
  - 002—consultation rejected

Return Values

When the command finishes, the script receives an ev_consultation_done.

Example

```
leg consult response leg_incoming -i $tcl_consultid
leg consult response $xorCallLeg -t $newTargetNum
leg consult response leg_incoming -c 2
```

Command Completion

Immediate

Related Events

ev_consult_request

leg consult request

This command is used to send a call-transfer consultation identifier request on the specified leg. Depending on the underlying protocol, the gateway will send a message to the endpoint or the gateway itself generates the identifier.

Syntax

leg consult request *legID*

Arguments

legID—The ID of the call-leg to transfer-target endpoint.

Return Values

None

Example

```
leg consult request $targetleg
```

Command Completion

When the command finishes, the script receives an ev_consult_response.

Related Events

ev_consult_response

leg disconnect

The **leg disconnect** command disconnects one or more call legs that are not part of any connection.

Syntax

leg disconnect {*legID* | *info-tag*} [-c *cause_code*] [-g <*gtd_handle*>] [-i <*iec*>]

Arguments

- *legID*—ID of the call leg.
- info-tag—A call leg info-tag that maps into one or more call legs. For more information about info-tags, see Chapter 4, "Information Tags."
- *-c cause_code*—An integer ISDN cause code for the disconnect. It is of the form di-*xxx* or just *xxx*, where *xxx* is the ISDN cause code.

Note Tcl IVR does not validate cause_code. For non-DID calls, the optional cause_code parameter does not have any effect on incoming telephony legs when both of the following conditions are true:

1. The **leg setupack** command has been issued for this leg.
2. The leg has not yet reached the connect state.

In this case, the cause_code parameter is ignored and the leg is disconnected using cause code 0x10, "Normal Call Clearing."

- -g <*gtd_handle*>—The handle to a previously-created GTD area.
- -i <*iec*>—Specifies an Internal Error Code (IEC) to be logged as the reason for the disconnect. See set iec, page 4-45, for possible values.

Return Values

None

Command Completion

When the command finishes, the script receives an ev_disconnect_done event.

Examples

```
leg disconnect leg_incoming
leg disconnect leg_outgoing
leg disconnect leg_all
leg disconnect 25
leg disconnect $callId
leg disconnect [info-tag get evt_legs]
leg disconnect leg_incoming -i media_done_err
leg disconnect leg_incoming 47 -i accounting_conn_err
```

Usage Notes

- If the specified call leg is invalid or if any of the specified call legs are part of a connection (conferenced), the script terminates with error output, and the call closes.
- When the script receives an ev_disconnected event, the script has 15 seconds to clear the leg with the **leg disconnect** command. After 15 seconds, a timer expires, the script is cleaned up, and an error message is displayed to the console. This avoids the situation where a script might not have cleared a leg after a disconnect.
- Using the *set iec* information tag in addition to specifying the IEC with the **leg disconnect** -<*iec*> command causes duplicate IECs to be associated with the call leg.

leg disconnect_prog_ind

The **leg disconnect_prog_ind** command sends a disconnect message with the specified progress indicator value to the specified leg.

Syntax

leg disconnect_prog_ind {*legID* | *info-tag*} [-c <*cause_code*>][-p <*prog_ind value*>]

Arguments

- *legID*—ID of the call leg.
- info-tag—A call leg info-tag that maps into one or more call legs. For more information about info-tags, see Chapter 4, "Information Tags."
- -c <*cause_code*>—An integer ISDN cause code for the disconnect. It is of the form di-*xxx* or just *xxx*, where *xxx* is the ISDN cause code.
- -p <*prog_ind value*>—The value of the call progress indication. Valid values are:
  - 1—PROG_NOT_END_TO_END_ISDN
  - 2—PROG_DEST_NON_ISDN
  - 4—PROG_RETURN_TO_ISDN
  - 8—PROG_INBAND
  - 10—PROG_DELAY_AT_DEST

Return Values

None

Command Completion

Immediate.

Examples

```
leg disconnect_prog_ind leg_incoming -c19 -p8
```

Usage Notes

- Applications that terminate a call can insert a leg disconnect_prog_ind before playing an announcement toward the incoming leg.
- This command is normally used on an incoming call leg before it reaches the connect state. Using this command on an outgoing call leg may result in an error or unexpected behavior from the terminating PSTN switch. Using this command on an incoming call leg that is already connected may result in an error or unexpected behavior from the originating PSTN switch.

leg facility

The **leg facility** command originates a facility message.

Syntax

leg facility {*legID* | *info-tag*} {*-s ss_Info* | *-g gtd_handle* | *-c*}

Arguments

- *legID*—The call leg ID the facility message is sent to.
- *info-tag*—A call leg info-tag that maps into one or more call legs. For more information about info-tags, see Chapter 4, "Information Tags."
- *-s ss_Info*—An array containing parameters that are passed to the stack to build the facility message.
- *-g gtd_handle*—Sends a new facility using the specified GTD handle.

- *-c*—Forwards the received facility message as is. Used when forwarding a received facility message to conferenced call legs. The raw message in the previous facility message is copied to the new facility message and updated.

Return Values

None

Command Completion

Immediate

Examples

```
set ssInfo (ssID) "ss_mcid"
leg facility leg_incoming -s ssInfo

object create gtd gtd_inr INR
object append gtd gtd_inr iri.1.inf 1
leg facility leg_incoming -g gtd_inr
```

Usage Notes

One of the following options is mandatory: *-s ss_info*, or *-g gtd_handle*, or *-c*

If the -s *ss_Info* option is used, a mandatory parameter, ssID, must be set to indicate the service type. The value for malicious call identification (MCID) messages is *ss_mcid*.

leg proceeding

The **leg proceeding** command sends a call proceeding message to the incoming call leg. The gateway is responsible for translating this message into the appropriate protocol message (depending on the call leg) and sending them to the caller.

Syntax

leg proceeding {*legID* | *info-tag*}

Arguments

- *legID*—The ID of the incoming call leg.
- *info-tag*—A call leg info-tag that maps into one or more call legs. For more information about info-tags, see Chapter 4, "Information Tags."

Return Values

None

Command Completion

Immediate

Example

```
leg proceeding leg_incoming
```

Usage Notes

- If the specified call leg is not incoming, this command clears the call.

- If leg_incoming is specified and there is more than one incoming call leg, a call proceeding message is sent to all the incoming call legs that have not already received a call preceding message.
- If the state of the specified call leg prevents it from receiving a call proceeding message (for example, if the state of the call leg is disconnecting) the command fails.
- If a call proceeding message has already been sent, this command is ignored. If IVR debugging is on (see the "Testing and Debugging Your Script" section on page 2-8), the command that has been ignored is displayed.
- If the specified call leg is already in the connect state, the script terminates and displays an error to the console, and the call is cleared.

leg progress

Sends a progress message to the specified leg.

Syntax

leg progress *{legID | info-tag}* [**-p** <*prog_ind_value*>] [**-s** <*sig_ind_value*>] [**-g** <*GTDHandle*>]

Arguments

- *legID | info-tag*—Points to the incoming leg to send the progress message to.
- **-s** <*sig_ind_value*>—The value of the call signal indication. The value is forwarded as is.
- **-p** <*prog_ind_value*>—The value of the call progress indication. Valid values are:
  - 1 (PROG_NOT_END_TO_END_ISDN)
  - 2 (PROG_DEST_NON_ISDN)
  - 4 (PROG_RETURN_TO_ISDN)
  - 8 (PROG_INBAND)
  - 10 (PROG_DELAY_AT_DEST)
- **-g** <*GTD handle*>—The handle to a previously created GTD area.

Return Values

None.

Command Completion

Immediate.

Examples

```
leg progress leg_incoming -p 8 -g gtd_progress_handle
```

Usage Notes

- Applications that terminate a call can insert a leg progress before playing an announcement toward the incoming leg.
- If the specified call leg is already in the connect state, the script terminates and displays an error to the console, and the call is cleared.

Note For incoming ISDN call legs, a setupack, proceeding, or alert message must be sent before the connect message. Otherwise, the script will receive an ev_disconnected event and the incoming leg will be disconnected.

leg senddigit

Transmits a digit on the specified call leg.

Syntax

leg senddigit {*legID* | *info-tag*} digit [*-t duration*]

Arguments

- *legID*—The ID of the call leg on which to send a digit.
- *info-tag*—A direct mapped info-tag mapping to one or more legs. For more information about info-tags, see Chapter 4, "Information Tags."
- *digit*— Specifies a single digit {0-9, A-D, *, #}
- *-t duration*— Specifies the duration of the digit in milliseconds.

Return Values

None

Example:

```
set digit 5
set duration 55
leg senddigit leg_outgoing $digit
or
leg senddigit leg_outgoing $digit -t $duration
```

Usage Notes

- Only a single digit can be specified for the **leg senddigit** verb. If more than one digit is specified, a syntax error is generated. The script terminates and displays an error message on the console, and call is cleared.
- The specified digit must be either 0 to 9, A to D, *, or #, otherwise the digit is not transmitted and a debug message is printed.
- The default digit duration is 100 ms. If the digit duration is not specified, the default value is used.
- The minimum digit duration is 40 ms, and the maximum digit duration is 4 seconds. The maximum duration is approximately twice the duration required for the longpound (#). If the duration specified is less than 40 milliseconds or greater than 4 seconds, the digit duration is reset to the default value and a debug message is printed.
- The DTMF relay H245 alphanumeric mode of transportation does not transport digit duration. The digit duration is not used if it is specified in the TCL verb and the negotiated DTMF relay mode of transportation is H245 alphanumeric.
- Digit transmission fails if the **leg senddigit** verb is executed and the negotiated DTMF relay is either rtp-nte or cisco-rtp.
- In-band transmission of digits is not supported. Digit transmission fails if **leg senddigit** is executed and DTMF relay is not negotiated.

leg sendhookflash

Transmits a hook flash on the specified call leg.

Syntax

leg sendhookflash {*legID* | *info-tag*}

Arguments

- *legID*—The ID of the call leg on which to generate a hookflash.
- *info-tag*—A direct mapped info-tag mapping to one or more legs. For more information about info-tags, see Chapter 4, "Information Tags."

Return Values

None

Usage Notes

- A hook flash can be generated on IP call legs, FXO ports, and T1 CAS trunks if the signaling type and platform supports it.
- Restrictions:
  - Hook flash transmission fails if the DTMF relay is not negotiated, or if the negotiated DTMF relay is rtp-nte or cisco-rtp.
  - In-band transmission of hook flash is not supported.

leg setup

The **leg setup** command requests the system to place a call to the specified destination numbers.

Syntax

leg setup {*destination* | *array-of-destinations*} *callinfo* [*legID* | *info-tag*] [**-g** <*GTDHandle*>] [**–d** <*dialpeerHandle*>]

Arguments

- *destination*—The call destination number.
- *array-of-destinations*—An array containing multiple call destination numbers.
- *callinfo*—An array containing parameters that determine how the call is placed. See the **set callinfo** command for possible values.
- *legID*—The call leg ID to conference if the call setup succeeds. For call transfer, this is usually the call leg that was conferenced with the leg that received the **ev_transfer_request** event. This leg should not be part of any conference.
- *info-tag*—A direct mapped info-tag mapping to one incoming leg. For more information about info-tags, see the "Information Tags" section on page 4-1.
- **-g** <*GTD handle*>—The handle to a previously created GTD area.
- **-d** <*dialpeerHandle*>—Specifies the dial-peer handle to use for the setup.

Return Value

None

Command Completion

When the command finishes, the script receives an **ev_setup_done** event.

Example

```
set callInfo(alertTime) 25
leg setup 9857625 callInfo leg_incoming
set destinations(1) 9787659
set destinations(2) 2621336
leg setup destinations callInfo leg_incoming

set dest leg_outgoing
set dialpeer_handle new_handle
leg setup $dest callInfo -d $dialpeer_handle

set setupSignal(Subject) "Hotel Reservation"
set setupSignal(Priority) "urgent"
set setupSignal(X-ReferenceNumber) "1234567890"
set callInfo(protoHeaders) setupSignal
set destination "4085551234"
leg setup destination callInfo leg_incoming
```

Usage Notes

- If the specified call leg is invalid, the script terminates and displays an error to the console, and the call is cleared.
- If a single destination number is specified, the **leg setup** command places a call to that destination number. When the destination phone rings, the incoming call leg is alerted (in-band or out-of-band, as appropriate). When the destination phone is answered, the call is connected, and the **leg setup** command returns an ev_setup_done event. If the call fails to reach its destination through the dial peer, the **leg setup** command tries the next dial peer until all dial peers that match the destination have been tried. (This is called *rotary hunting*.) At that point, the **leg setup** command fails with a failure code (an ev_setup_done event with a status code of alert timeout). For more information about the failure codes, see the "Status Codes" section on page 5-6.
- If multiple destination numbers are specified, the **leg setup** command places the call to all the specified numbers simultaneously (causing all the destination phones to ring at the same time). When the first destination phone is answered, the call is connected and the remaining calls are disconnected. (This is called *blast calling*.) Therefore, when you receive the ev_setup_done event and then issue an **infotag get evt_legs** info-tag command, the incoming leg is returned.
- A script can initiate more than one **leg setup** command, each for a different call leg ID. After a call setup message has been issued for a specific call leg ID, you cannot issue another **leg setup** command for this call leg ID until the first one finishes.
- If a prompt is playing on the call leg when the call setup is issued, the leg setup proceeds and the destination phones ring. However, the caller does not hear the ring tone until the prompt has finished playing. If, during the prompt, the destination phone is answered, the prompt is terminated and the call is completed.
- If the specified call leg is invalid, the script terminates and displays an error to the console, and the call is cleared.
- The leg ID used in the **leg setup** command should not be conferenced. Otherwise, the command fails and the script terminates.
- If successful, this command returns the following:

  - *legID*—The unique IDs assigned to the two legs that are part of the connection. The ID of the incoming leg might not be what you passed as the incoming leg. The incoming leg might have been cleared and a new incoming leg conferenced. This is an exception case that might happen because of supplementary services processing or H.450 services.
  - *connectionID*—A unique ID assigned to this connection. This ID is required for the **connection destroy** command.

  The above information can be obtained from evt_legs and evt_connections info-tags. For more information about info-tags, see the "Information Tags" section on page 4-1.

  If unsuccessful, this command returns nothing or a single leg ID. You may get the incoming leg ID because the incoming leg that was passed may have been disconnected. These are exception cases that may happen due to supplementary services processing or H.450 services.

- The script can terminate a pending call setup by issuing the **command terminate** verb. See the **command terminate** section for more information.
- Leg setup cannot use a leg that has a dialog running in its [*legID* | *info-tag*] parameter.
- The [*legID* | *info-tag*] is an optional parameter. Tcl IVR applications can initiate a leg setup without referencing an incoming leg. This ability can be useful in applications such as a callback application. After the leg setup successfully completes, the application can connect the new leg with an existing leg using the **connection create** verb.
- If there is no destinationNum, then <destination> is used for outbound dial-peer selection. If both destinationNum and <destination> exist, then <destination> is used to select the outbound dial peer, but destinationNum is used to fill out the signaling fields.
- If the destinationNum and originationNum contain a URL, the application extracts the E.164 from the URL, if any, and stores it directly into the calledNumber and callingNumber fields, respectively. Otherwise, they work as normal. These fields take only E.164 or sip:/tel: URLs. If any other URL format is used, the application throws an Unsupported Format error.
- Passing and accessing SIP message bodies is not supported.
- An example of how multiple SIP headers are set in Tcl is as follows:

```
set <array_name_xxx>(<header name>) <"header value">
set <array_name_xxx>(<another header name>) <"header value">
...
set callinfo(protoHeaders) <array_name_xxx>
```

  For example, if we wanted to set the following headers to be sent in the call setup:

```
From = abc@xyz.com
To = joe@big.com
Subject = "Hello"
```

  we could do this in a Tcl script as follows:

```
# The array name "headers" can be any name you want
set headers (From) "blah@xyz.com"
set headers (To) "joe@big.com"
set headers (Subject) "Hello"
# Here, we set the array "headers" in the callInfo array, mimicking a
two-dimensional array
set callInfo(headers) headers
```

  We then send them in the call setup, as follows:

```
leg setup $dest callinfo leg_incoming
```

leg setup_continue

The **leg setup_continue** command allows the application to interact with the system during setup. This command is used to initiate a setup to an endpoint address or to let the system continue its action after an event interrupts the call processing. Typically, the application uses this verb after it receives the result of the address resolution or a call signal.

Note The application can stop the leg setup by using the '**handler terminate'** verb.

Syntax

leg setup_continue <*command handle*> [**-a** <*endpointAddress* | *next*>] **[-d** <*dialpeerHandle*>] [**-g** <*GTDHandle*>] **[-c** <*callInfo*>]

Arguments

- *command handle*—The command handler received from the **get evt_last_event_handle** information tag.
- -**a** <*endpointAddress*|*next*>—Indicates to the system to initiate the setup with a particular endpoint address or the next endpoint address. The initial address is typically the primary endpoint address. If the application specifies 'next' after it receives the address resolution results, the first (primary) endpoint address is used.
- -**d** <*dialpeerHandle*>—Specifies the dial peer handle to use for the setup.
- **-g** <*GTD handle*>—The handle to a previously created GTD area.
- -**c** <*callInfo*>—If this optional parameter is used, the application passes the callInfo array for use in the endpoint setup. The following parameters can be updated on a per-endpoint setup basis:
  - originationNum
  - originationNumToN
  - originationNumPI
  - originationNumSI

  See set callinfo, page 3-72, for more information.

Return Values

None.

Command Completion

If the command is used to initiate the setup to an endpoint address, when it finishes, the script receives an **ev_setup_done** event if successful or an **ev_disconnect** if the setup fails.

If the command is used to let the system continue its action after an event interrupts the call processing, it finishes immediately.

Examples

```
leg setup_continue $commandHandle -a next -g gtd_alert_handle
```

Usage Notes

- To retrieve the command handle associated with the leg setup, the application can use the infotag **get evt_last_event_handle**.
- The leg setup_continue should not be used if the address resolution fails with a status code other than ar_000. In such cases, the application may issue a new leg setup command with another dial peer.
- Other fields of the callInfo structure, if set, are ignored.
- New callInfo parameter values will continue to be used for subsequent endpoint setups until changed.
- To continue the call setup after intercepting the *ev_address_resolved* event, *-a <endpointAddress | next>* should be specified. When only the command handle is specified to **leg setup_continue**, the system assumes you are continuing the call setup after intercepting a backward signaling event.

leg setupack

The **leg setupack** command sends a setup acknowledgement message on the specified incoming call leg.

Note The ISDN state machine actually connects the incoming call on a setup acknowledgement.

Syntax

leg setupack {*legID* | *info-tag*}

Arguments

- *legID*—The ID of the call leg to be handed off.
- *info-tag*—A call leg info-tag that maps to one or more incoming legs. For more information about info-tags, see Chapter 4, "Information Tags."

Return Values

None

Command Completion

Immediate

Example

```
leg setupack leg_incoming
```

Usage Notes

- The **leg setupack** command can be used only once in a Tcl IVR application. Any application that executes this command more than once will abort.
- If the specified call leg is not an incoming call leg, this command clears the call.
- If leg_incoming is specified and there are multiple incoming call legs, a setup acknowledgement is sent to all the call legs that have not been previously acknowledged.
- When the **leg setupack** command is applied to an incoming ISDN call leg, the underlying ISDN protocol stack sends a *proceeding* message followed by a *connect* messge to the originating ISDN switch. This is done to establish the voice path so the voice application is able to collect digits.

- The specified call leg must be in the initial call state. If a setupack, proceeding, progress, alerting, or connect messsage has already been sent on the specified call leg, the script terminates and displays an error to the console, and the call is cleared.

leg tonedetect

The **leg tonedetect** command enables or disables the detection of specific tones during a call.

If tone detection is enabled and a tone is detected, an ev_tone_detected event is generated. This event is generated only after a required minimum time has elapsed, as determined by <Number Cycles>. At most, one event is generated per tone type requested. If an enable command is issued again for a tone type that is already being detected, that tone type is reenabled.

Syntax

```
leg tonedetect {legID | info-tag} enable {tonetype} [<Number Cycles>]
leg tonedetect {legID | info-tag} disable <{tonetype}> <{ignoremintime}>
```

Arguments

- legID—The ID of the call leg
- *info-tag*—A call leg info-tag that maps to one or more incoming legs. For more information about info-tags, see Chapter 4, "Information Tags."
- *tonetype*—The type of tone to detect.
  - Possible value: cng (a series of CNG tones)
- *Number Cycles*—The number of consecutive single tone cycles required before ev_tone_detected is generated. If this argument is not specified, the default value is 1 cycle.
- *ignoremintime*—Suppress messages that may warn that insufficient time was allowed for tone detection.

Return Values

- For enable:
  - A string indicating the period (in seconds) required for the tone to be detected and for the event to be generated or a string indicating error. The required minimum time is computed by Number Cycles times the minimum time required for that specific tone.
- For disable:
  - Tcl_OK or Tcl_ERROR. Error occurs when this command is called in less than the minimum time required and when *ignoremintime* is not specified. For example, if the required minimum time is 7 seconds and this command is called after 3 seconds, the tone detection can only be disabled if *ignoremintime* is specified.)

Example

```
set MIN_CNG_DETECTION_TIME [leg tonedetect leg_incoming enable cng]
leg tonedetect leg_incoming disable cng ignoremintime
```

Command Completion

None

Usage Notes

None

leg transferdone

This command indicates the status of the call transfer on a call-leg and, depending on the status, may send a disconnect or facility message to the call leg.

Syntax

leg transferdone *{legID | info-tag} transferStatusCode*

Arguments

- *legID*—The ID of the call-leg
- *transferStatusCode*—Success/Failure. See Transfer Status, page 5-13, for a list of possible values.

Return Values

The command returns one of the following:

- *0 (success)*—Success
- *1 (failed, unsupported)*—The signaling protocol associated with the specified leg is not capable of carrying this information. This will not trigger a script error.

Example

```
leg transferdone leg_incoming ts_011
set retcode [leg transferdone leg_incoming ts_000]
```

Command Completion

For a success return value, the command finishes by sending ev_disconnected to the script.

Usage Notes

If the specified call leg is invalid for this operation, the script terminates with error output, and the call closes.

leg vxmldialog

The **leg vxmldialog** command initiates a VoiceXML dialog on the specified leg. The markup for the dialog to be directed at the leg is specified either by a URI or by an actual markup as a string parameter. The script can also pass a list of variables as parameters. These variables are available, by copy, to the VoiceXML dialog session.

When a VoiceXML dialog is active on a leg, no other operations or commands are permitted on the leg except for the **command terminate** and **leg vxmlsend** commands. If the VoiceXML dialog completes or terminates, either normally or abnormally, an *ev_vxmldialog_done* event will be received by the script and an appropriate status code, indicating the reason for termination, can be retrieved through the *evt_status* information tag.

If both the *-u* and *-v* options are specified, the inline VoiceXML dialog executes in the *-v* option and uses the *-u* URI as the default base URI as if the inline code was downloaded from there. A VoiceXML dialog refers the entire VoiceXML session that is initiated on a leg by a **leg vxmldialog** command, starting with an initial inline document or URI, and may span through multiple documents during the course of the conversation.

Initiating a VoiceXML dialog segment on individual call legs from within a Tcl application is called *hybrid scripting*. Hybrid scripting differs from the concept of application handoff, where the call leg is completed and handed off to another application, then loses control of the leg. For more information on call handoff, see the "Call Handoff in Tcl" section on page 1-5. For more information on hybrid scripting, see the "Tcl/VoiceXML Hybrid Applications" section on page 1-6.

Syntax

leg vxmldialog <*legID*> -u <*dialog-uri*> *[*-p <*param-array*>*] [*-v <*dialog-markup-string*>*]*

Arguments

- *legID*—The ID of the call leg to be handed off.
- *dialog-uri*—A URI to retrieve the dialog markup from or to use as a base URI when used with the *-v* option.
- *param-array*—A Tcl array containing the list of parameters to pass to the dialog markup. The VoiceXML session can access these parameters through session variables of the form *com.cisco.params.xxxxxx*, where *xxxxxx* was the index in the Tcl array array. The values of the Tcl array variables will be available to the VoiceXML application as text strings. The only exception to this rule is when a Tcl array variable contains memory *ram://URI,* pointing to an audio clip in memory. In this case, the audio clip will be available to the VoiceXML document as an audio clip object.
- *dialog-markup-string*—A string containing the VoiceXML markup specifying the dialog to initiate on the leg.

Return Values

None

Command Completion

ev_vxmldialog_done

Example

```
leg vxmldialog leg_incoming
```

Usage Notes

- The VoiceXML dialog can be terminated using the **command terminate** command.
- When the dialog command is active on a leg, other Tcl IVR command operations, like **medial play**, **leg collectdigits**, and **leg setup**, are illegal. If these commands are executed, the application errors out and terminates as a Tcl IVR script error. The VoiceXML dialog also terminates.
- The <transfer> tag is not supported when VoiceXML is running in the dialog mode. If the VoiceXML dialog executes a <transfer> tag, an *error.unsupported.transfer* event is thrown to the VoiceXML interpreter.
- From a VoiceXML dialog, events can be sent to Tcl by usingthe com.cisco.ivr.script.sendevent object. For more information on sendevent objects, see SendEvent Object, page 1-8.

leg vxmlsend

The **leg vxmlsend** command throws an event at an ongoing VoiceXML dialog on the leg. The event thrown to the VoiceXML dialog is of the form <*event-name*>. The event can carry parameters associated with it and are specified by <*param-array*>. The Tcl associative array contains the list of parameters to

send to the dialog along with the event. The index of the array is the name of the parameter as accessible from the VoiceXML dialog and the value is the value of the parameter as accessible from the VoiceXML dialog.

These parameters are available to the VoiceXML script through the *variable_message* and is an object containing all the Tcl array indexes as subelements of the message object. If there is not a VoiceXML dialog executing on the leg, this command simply succeeds and is ignored.

Syntax

leg vxmlsend <*legID*> <*event-name*> *[*-p <*param-array*>*]*

Arguments

- *legID*—The ID of the call leg to be handed off.
- *event-name*—Name of the event to throw to the VoiceXML dialog.
- *param-array*—A Tcl array containing a list of parameters to pass to the ongoing VoiceXML dialog. The VoiceXML session can access these parameters when the thrown VoiceXML event is caught in a catch handler. The parameters are accessible through the *_message.params.xxxxxx* variable, which is catch-handler scoped and therefore available within the catch handler. The values of the Tcl array variables are available to the VoiceXML application as text strings. The only exception to this rule is when a Tcl array variable contains memory *ram:// URI* pointing to an audio clip in memory. In this case the audio clip is available to the VoiceXML document as an audio clip object to the VoiceXML document.

Return Values

None

Command Completion

Immediate

Example

```
leg vxmlsend leg_incoming $event-name
```

Usage Notes

None

log

The **log** command originates a syslog message.

Syntax

log *-s <CRIT | ERR | WARN | INFO> <message text>*

Arguments

- *-s <CRIT | ERR | WARN | INFO>*—The severity of the message.
  - CRIT—Critical
  - ERR—Error message (default)
  - WARN—Warning message
  - INFO—Informational message

- *message text*—The body of the message. Use double quotes or braces to enclose text containing spaces or special characters.

Return Values

None

Command Completion

Immediate

Examples

```
set msgStr "MCID request succeeded"
append msgStr [clock format [clock seconds]]
log $msgStr
```

Usage Notes

- The **log** command uses the Cisco IOS message facility to send the message. Except for critical messages, rate limitations are applied to the emission of IVR application log messages. The minimum time intervals between emissions of the same message are as follows:
  - ERR—1 second
  - WARN—5 seconds
  - INFO—30 seconds

  A message is considered the same if the application issues a **log** command with the same severity.
- When performing the rate-limitation, the Cisco IOS message facility takes the emissions of all IVR applications into consideration. If a message cannot tolerate the rate limitation, use the CRIT severity level.
- The message text should be as clear and accurate as possible. The operator should be able to tell from the message what action should be taken.
- The system appends a new line character after the message, so there is no need to use a new line character.
- Use the **log** message facility to report errors. Use the **puts** command for debugging purpose.
- Log messages can be sent to a buffer, to another TTY, or to logging servers on another system. See the Cisco IOS Troubleshooting and Fault Management logging command for configuration options.
- Sending a large number of log messages to the console can severely degrade system performance. Log messages sent to the console may be suppressed by the **logging console <level>** CLI command. Alternatively, the console output can be rate-limited by using the **logging rate-limit console** CLI command. To disable logging to the console altogether, especially if logging is already directed to a buffer or a syslog server, use the **no logging console** command.

media pause

The **media pause** command temporarily pauses the prompt that is currently playing on the specified call leg.

Syntax

media pause {*legID* | *info-tag*}

Arguments

- *legID*—The ID of the call leg to which to pause play of the prompt.
- *info-tag*— A direct mapped info-tag mapping to one or more legs. For more information about info-tags, see Chapter 4, "Information Tags."

Return Values

None

Command Completion

This command has immediate completion. However, the script should be prepared to receive an ev_media_done event if the command fails. An ev_media_done event is not generated when this command is successful.

Example

```
media pause $legID
```

Usage Notes

If the specified call leg is invalid, the script terminates and displays an error to the console, and the call is cleared.

media play

The **media play** command plays the specified prompt on the specified call leg.

Syntax

media play {*legID* | *info-tag*} {<*url*> | <*token*>}+

Arguments

- *legID*—The ID of the call leg to which to play the prompt.
- *info-tag*— A direct mapped info-tag mapping to exactly one leg. For more information about info-tags, see Chapter 4, "Information Tags.".
- *url*—The URLs of the prompts to be played. The value of *url-list* can be a list of URLs for individual prompts or a list of strings, each of which is a collection of URLs. The URL can point to a prompt from Flash memory, an FTP server, a TFTP server, or an RTSP prompt. The strings could be dynamic prompts, in which case they are strings that describe the dynamic prompt using a special notation format to specify what to play and in what language. See "Usage Notes" below.
- *token*—Returned from the HTTP command, **geturl**, or the **media record** command, *token* points to a recording that will be played directly from RAM. When *token* points to any other recording url, the url is used to fetch the audio.

Note The media content created from playing the recording is not cached.

In order to use the *token* returned from **geturl**, the content type should be "*audio/**". If the *statearray* associated with the token has a codec element defined, the body is treated as raw, or headerless, audio in the specified codec. If there is no codec element defined, the body is parsed to match either a .au or .wav file. If it does not contain a .au or .wav header, the media play fails.

- @C<string>—Plays out the alphanumeric characters one by one. For example, @Ccsco will play "C" "S" "C" "O". The supported inputs are the printable ASCII character set.
- %Wday_of_week—Plays out the day of week prompt. For example, %w1 will play "Monday". The values 1–7 represent Monday to Sunday.
- %Ttime_of_day—Accepts an ISO standard time format and plays out the time. For example, %T131501 will play "one" "fifteen" "pm" "one" "second". Supported formats are: hhmmss, hhmm and hh, where hh is hour, mm is minute and ss is second. Hour is in 24-hour format.
- %Ddate—Accepts an ISO standard date format and plays out the date. Supported formats are: CCYYMMDD, CCYYMM, CCYY, --MMDD, --MM or ---DD, where CC is century, YY is year, MM is month and DD is day of month. For example, %D20000914 will play "year" "two" "thousand" "september" "fourteenth"; %D199910 will play "year" "nineteen" "ninety" "nine" "october"; %D2001 will play "year" "two" "thousand" "one"; %D--0102 will play "January" "second"; %D--12 will play "december"; and %D---31 will play "thirty" "first".

Return Values

None

Command Completion

When the **media play** command completes, the script receives an *ev_synthesizer* event instead of an *ev_media_done* event. For backward compatibility the gateway still supports *ev_media_done* events, but going forward its encouraged to use the *ev_synthesizer* event for detection of play completion.

Examples

```
media play leg_incoming@C$alpha
media play leg_incoming@C$ascii
media play leg_incoming@C\ !\"#\$%&'()*+,-./0123456789:\;<=>?@\[\\]^_`{|}~
media play leg_incoming%D2001
media play leg_incoming%D201211
media play leg_incoming%D20300830

media play leg_incoming%D---01 %D---02 %D---03 %D---04 %D---05 %D---06 %D---07 %D---08
%D---09 %D---10 %D---11 %D---12 %D---13 %D---14 %D---15 %D---16 %D---17 %D---18 %D---19
%D---20 %D---30

media play leg_incoming%D---21 %D---22 %D---23 %D---24 %D---25 %D---26 %D---27 %D---28
%D---29 %D---31

media play leg_incoming%T01 %T02 %T03 %T04 %T05 %T06 %T07 %T08 %T09 %T10 %T11 %T12 %T13
%T14 %T15 %T16 %T17 %T18 %T19 %T20 %T21 %T22 %T23 %T00

media play leg_incoming%T24
media play leg_incoming%W1 %W2 %W3 %W4 %W5 %W6 %W7

set audio_file http://prompt-server1/prompts/en_welcome.au
media play leg_incoming $audio_file
```

Usage Notes

- If a prompt is already playing when the **media play** command is issued, the first prompt is terminated and the second prompt is played.
- The **media play** command takes a list of URLs or prompts and plays them in sequence to form a single prompt. The individual components of the prompt can be full URLs or Text-to-Speech (TTS) notations. The possible components of the prompt are as follows:
  - URL—The location of an audio file. The URL must contain a colon. Otherwise, the code treats it as a file name, and adds .au to the location.
  - *name*.au—The name of an audio file. The currently active language and the audio file location values are appended to the *name*.au. The filename cannot contain a colon, or it is treated as a URL.
  - %a*num*—A monetary amount (in US cents). If you specify 123, the value is $1.23. The maximum value is 99999999 for $999,999.99.
  - %t*num*—Time (in seconds). The maximum value is 999999999 for 277,777 hours 46 minutes and 39 seconds.
  - %d*day_time*—Day of week and time of day. The format is DHHMM, where D is the day of week and 1=Monday, 7=Sunday. For example, %d52147 plays “Friday, 9:47 PM.”
  - %s*time*—Amount of play silence (in ms).
  - %p*num*—Plays a phone number. The maximum number of digits is 64. This does not insert any text, such as “the number is,” but it does put pauses between groups of numbers. It assumes groupings as used in common numbering plans. For example, 18059613641 is read as 1 805 961 3641. The pauses between the groupings are 500 ms.
  - %n*num*—Plays a string of digits without pauses.
  - %i*id*—Plays an announcement. The *id* must be two digits. The digits can be any character except a period (.). The URL for the announcement is created as with _announce_<id>.au, and appending language and au location fields.
  - %clanguage-index—Language to be used for the rest of the prompt. This changes the language for the rest of the prompts in the current **media play** command. It does not change the language for the next **media play** command, nor does it change the active language.
- If no argument is given to the TTS notation, the notation is ignored by IVR; no error is reported.
- **Media play** with a NULL argument for %c uses the default language for playing prompts, if there are valid prompts, along with a NULL %c. Previously, the script would abort.
- If the specified call leg is invalid, the script terminates and displays an error to the console, and the call is cleared.
- If the call leg specified by an information tag maps to more than one leg, the script terminates, sends an error message to the console, and clears the call. The use of *leg_all* is not recommended, since this is more likely to map to multiple legs.
- The **media play** command cannot be applied to a leg that is part of a connection. When executed to a conferenced leg, the script aborts with message "Leg is in Conferenced state". The connection must be destroyed, then the media play can run and the connection can be re-created.
- Multi-language support through Tcl-based language scripts must be enabled in order to use the newly defined dynamic prompts: @Ccharacters, %Wday_of_week, %Ttime_of_day, and %Ddate. See the command **call language voice** in the *Enhanced Multi-Language Support for Cisco IOS Interactive Voice Response* document. Only the English version of these new dynamic prompts are supported.

media record

The **media record** command records the audio received on the specified call leg and saves it to the location specified by the URL.

Syntax

media record {*legID* | *info-tag*} [-p <*recordInfo*>] [<*url*>]

Arguments

- *legID*—The ID of the call leg whose audio will be recorded.
- *info-tag*—A direct-mapped info-tag mapping to exactly one leg. For more information about info-tags, see Chapter 4, "Information Tags."
- -p <*recordInfo*>—A Tcl array containing any of the following:
  - *codec*—An integer value used to specify codec to be used during recording. The following are possible values:

    2—voipCodecG726r16

    3—voipCodecG726r24

    4—voipCodecG726r32

    5—voipCodecG711ulaw

    6—voipCodecG711Alaw

    7—voipCodecG728

    8—voipCodecG723r63

    9—voipCodecG723r53

    10—voipCodecGSM

    11—voipCodecGSMefr

    12—voipCodecG729b

    13—voipCodecG729ab

    14—voipCodecG723ar63

    15—voipCodecG723ar53

    16—voipCodecG729IETF
  - *finalSilence*—Finalsilence specified in milliseconds, where 0 indicates no finalsilence.

    range: 0–MAXINT

    default: 0
  - *dtmfTerm*—Terminate the record with a DTMF key.

    enable: Enables terminating the record with a DTMF key

    disable: Disables terminating the record with a DTMF key

    default: Enable

- *maxDuration*—Specifies the maximum duration, in milliseconds, allowed for recording, where 0 indicates the recording will terminated by the configured limit.

  range: 0–MAXINT

  default: 0

- *maxMemory*—Specifies the maximum memory allowed for this recording in bytes, where 0 indicates the recording will be terminated by the configured session limit.

  range: 0–MAXINT

  default: 0

- *fileFormat*—Allowed file format. Possible values are:

  au: Sun file format

  wav: wav file format

  none: raw audio (no file header will be attached)

  default: au

- *beep*—Play a beep before recording. Possible values are:

  nobeep: do not play a beep before recording

  beep: play a beep before recording

  default: nobeep

Note Any of the above values that are not valid results in the failure of the *media record* verb.

- *url*—The location of the target file that the audio will be recorded to. The following are possible values:
  - rtsp—Records audio to the rtsp server if the rtsp server supports recording.
  - tftp—Records to the tftp server.
  - flash—Records to flash.
  - http—Records to an http server.
  - ram—Records to memory.

  If no url is specified, ram recording is assumed. The returned token represents the recording. The token can be used to playback or to get information about the recording.

Return Values

A token describing the media content created for the recording, which can be used as a Tcl array to get information about the recording. Use the following construct to create an easy-to-use array variable:

```
upvar #0 $token myrecording
```

The following elements of the array are returned.

- *url*—Contains the url of the recording.
- *duration*—Stores the length of the recording in milliseconds.
- *totalsize*—Contains the size of the recording in bytes.

- *type*—Contains the content type of the audio file. Possible values include:
  - audio/basic
  - audio/wav
- body—The pointer to the actual voice data.

Command Completion

The script receives an *ev_recorder* event when the recording terminates after the specified duration, after the application issues a **media stop** command, or if terminated by DTMF. If the recording is terminated by a **leg disconnect** command, the script does not receive an *ev_media_done* event; it receives only an *ev_disconnected* event for the leg. If the recording is successful, it can be accessed at the location specified in the URL.

The status of the recording can be accessed using **infotag get** *evt_status* after receiving the *ev_media_done* event.

Media seek and media pause does not affect a media recording.

Possible values are:

- ms_101—Failure to record.
- ms_103—Invalid URL.
- ms_105—Recording stopped due to dtmfKey termination.
- ms_106—Recording stopped because MaxTime allowed is reached.
- ms_107—Recording stopped because MaxMemory allowed is reached.
- ms_109—Recording stopped because of a silence timeout.

Example

```
set recordInfo(codec) g711ulaw
set recordInfo(finalSilence) 0
set recordInfo(dtmfTerm) enable
set recordInfo(maxDuration) 5000
set recordInfo(fileFormat) au
set recordInfo(beep) nobeep
set url ram
media record leg_incoming $recordInfo $url
```

Usage Notes

- If the specified call leg is invalid, the script terminates, displays an error on the console, and clears the call.
- If the call leg specified by an information tag maps to more than one leg, the script terminates, displays an error on the console, and clears the call. The use of leg_all is not recommended, since this is more likely to map to multiple legs.
- If the specified call leg is already being recorded, the script receives an *ev_media_done* event indicating a failure for the second media record invocation. The script receives another *ev_media_done* event when the first recording completes.
- It is okay for the specified call leg to be in the conferenced state. In this case, only the audio received from the specified leg is recorded.
- Simultaneous playout and record on a single call leg is not supported. Attempts to do this may result in unexpected or undesirable behavior.

media resume

The **media resume** command resumes play of the prompt that is currently paused on the specified call leg.

Syntax

media resume {*legID* | *info-tag*}

Arguments

- *legID*—The ID of the call leg to which to resume play of the prompt.
- *info-tag*— A direct mapped info-tag mapping to one or more legs. For more information about info-tags, see Chapter 4, "Information Tags."

Return Values

None

Command Completion

This command has immediate completion. However, the script should be prepared to receive an ev_media_done event if the command fails. An ev_media_done event is not generated when this command is successful.

Example

```
media resume $legID
```

Usage Notes

If the specified call leg is invalid, the script terminates and displays an error to the console, and the call is cleared.

media seek

The **media seek** command does a relative seek on the prompt that is currently playing. This command moves the prompt forward the specified number of seconds within the message.

Syntax

media seek {*legID* | *info-tag*} *time-in-seconds*

Arguments

- *legID*—The ID of the call leg.
- *info-tag*— A direct mapped info-tag mapping to one or more legs. For more information about info-tags, see Chapter 4, "Information Tags."
- *time-in-seconds*—The number of seconds to seek forward. If you specify a negative number, the prompt moves backward in the message.

Return Values

None

Command Completion

This command has immediate completion. However, the script should be prepared to receive an ev_media_done event if the command fails. An ev_media_done event is not generated when this command is successful.

Example

```
media seek $legID +25
media seek $legID -10
```

Usage Notes

- If the specified call leg is invalid, the script terminates and displays an error to the console, and the call is cleared.
- This command works only with RTSP prompts. If there are non-RTSP-based prompts on the prompt list that is currently playing, the command does not work.
- If you specify a number of seconds greater than the remaining time in the prompt, the seek moves to the end of the prompt and the script receives an ev_media_done event.

media stop

The **media stop** command stops the prompt that is currently playing on the specified call leg.

Syntax

media stop {*legID* | *info-tag*}

Arguments

- *legID*—The ID of the call leg to which to stop the prompt.
- *info-tag*— A direct mapped info-tag mapping to one or more legs. For more information about info-tags, see Chapter 4, "Information Tags."

Return Values

None

Command Completion

Immediate. However, the script receives an ev_media_done event if the prompt completed before being stopped.

Example

```
media stop $legID
```

Usage Notes

If the specified call leg is invalid, the script terminates and displays an error to the console, and the call is cleared.

modulespace

The **modulespace** command allows the creation, access, and deletion of a modulespace in which a module can execute code.

Syntax

```
modulespace new arg
modulespace terminate arg
modulespace listen arg
modulespace unlisten arg
modulespace return final arg
modulespace return interim arg
modulespace event intercept arg
modulespace event consume arg
modulespace children module-handle
modulespace code script
modulespace current
modulespace eval module-handle arg arg
modulespace exists module-handle
modulespace inscope module-handle arg arg
modulespace parent
```

Arguments

- **modulespace new** <*context-string*>—Creates and installs a new submodule by creating a modulespace for it to run in. The new modulespace is created under the parent modulespace. This modulespace can execute a sub-state machine that is initialized by the existing **fsm define** command. The **modulespace new** command returns a *module-handle* to the newly-created modulespace. Creation of the modulespace also creates a variable namespace in the interpreter it is tied to. This variable namespace is active whenever the modulespace is active.

  The modulespace and its namespace is invoked by using the defined modulespace commands. These modulespace commands are very similar to the equivalent namespace command, with some limitations as noted in their sections. The value of *context-string* is available in all events generated by this module, such as the *ev_module_event* and the *ev_module_done* events. This context string could be used to provide some context information associated with this module, such as a data structure name or handle, or a call-back function to invoke for such events. This information can be accessed in the parent modulespace when processing a module event with the *evt_module_context* information tag.

- **modulespace terminate** <*module-handle*>—This command initiates the termination of an active modulespace. When the modulespace completes termination, its listeners will receive an *ev_module_done* event.

- **modulespace listen** <*leg-id* | *connection-id*>—This command adds a leg or connection object to the listen list of the modulespace. This means that all events associated with that leg or context are seen by this module before it is seen by the parent module. This allows the module to take action to implement its functionality and also decide whether the parent module should see this event or if it should be consumed or filtered from the parent. Note that when installing such modules to listen on objects, they are added in a specific order, and when the system receives an event for that object, the event is submitted for inspection to the modules on the listen list of the object one by one. Doing a listen on a leg that is already being listened by the current module is acceptable and is a no-op.

- **modulespace unlisten** <*leg-id* | *connection-id*>—This command does the opposite of the **modulespace listen** command. It removes the module from the listen list of the specified object. All events associated with the object will not be submitted to this module after this command is executed. Doing an unlisten on a leg that is not being listened by the current module is acceptable and is a no-op.

- **modulespace return final** <*param-array*>—This command results in the completion of the module execution, completing with an *ev_module_done* event to the parent module that invoked it. In the process, the module is removed from all objects currently listening and is added to the return list of

objects accessible by the parent module when it receives the *ev_module_done* event. These objects can be accessed by the parent module through the use of the *evt_legs* and *evt_connections* information tags.

This command also undefines or deletes the Tcl <*param-array*> variable or object from the current modulespace and passes along with it the *ev_module_done* event to the parent module. The information within <*param-array*> is accessible in the parent modulespace when handling the *ev_module_done* event by using the *evt_params* information tag, which creates an alias to the <*param-array*> information within the parent modulespace and makes it accessible from within the parent module. The module receiving the *ev_module_done* event then has access to the module handle that generated this event through the *evt_module_handle* information tag.

Note The **modulespace return final** command must be executed from within the modulespace of the module that is completing. Note that a module does not cleanup on its own unless orphaned. A module is classified as orphaned if it is not listening to any other objects or modules, and has no outstanding events such as AAA, timers, media commands, or HTTP requests. Also note that when a leg receives a disconnect event and has not been disconnected by the application within a certain time, the safety timer kicks in with a cleanup event that clears up the hung call and all modules, objects, and resources associated with it.

- **modulespace return interim** <*module-sub-event-name*> <*param-array*>—This command results in an intermediate *ev_module_event* event, which is generated by the module the command was executed in and received by the parent module that invoked the current module. The module receiving the *ev_module_event* event then has access to the module handle that generated the event through the *evt_module_handle* information tag. It also has access to the specific module subevent name through the *evt_module_subevent* information tag.

  The information within <*param-array*> is also accessible to the parent module when handling the *ev_module_event* event. The parent module can access this information by using the *evt_params* information tag, which can create an alias to the <*param-array*> information and make it accessible within the parent modulespace.

Note This command must be executed from within the modulespace of the module that wants to generate the interim event.

- **modulespace event intercept**—This command results in the event being intercepted by the current modulespace for its parent modulespace. The current event being processed by this module is submitted to the parent of the current module, even though it may not be listening to the object this event belongs to. In the absence of this command, the event is submitted to all modules that are listening to this object in the order in which they are listening.

  Note This command must be executed from within the modulespace of the module that is processing the current event and fails if it is in another modulespace. If not specified, the default is to continue.

- **modulespace event consume**—This command results in the event being consumed by the current module. The current event being processed is completed and freed, and is not submitted to other modules even though they may be listening to the object this event belongs to. Without this command, the event would be submitted to all modules listening to this object in the order they are listening.

  Note This command must be executed from within the modulespace of the module that is processing the current event and fails if it is in another modulespace. If not specified, the default is to continue.

- **modulespace children** *module-handle*—Returns a list of all child modulespace handles that belong to the modulespace *module-handle*. If *module-handle* is not specified, the children of the current modulespace are returned.
- **modulespace code** *script*—Captures the current modulespace context for later execution of the script. It returns a new script in which *script* has been wrapped in a **modulespace code** command. The new script has two important properties. First, it can be evaluated in any modulespace and causes *script* to be evaluated in the current modulespace (the one where the **modulespace code** command was invoked). Second, additional arguments can be appended to the resulting script and passed to *script* as additional arguments. For example, suppose the command **set script [modulespace code {foo bar}]** is invoked in modulespace *module-x*. Then **eval "$script x y"** can be executed in any modulespace, assuming the value of **script** has been passed properly, and will have the same effect as the command **modulespace eval module-x {foo bar x y}**. A scoped command captures a command together with its modulespace context in a way that allows it to be executed properly later.
- **modulespace current**—Returns the module handle for the current modulespace.
- **modulespace eval** *module-handle arg arg...*—Activates a modulespace referred to by *module-handle* and evaluates code in that context. If more than one *arg* argument is specified, the arguments are concatenated together with a space between each one in the same fashion as the **eval** command and the result is evaluated.
- **modulespace exists** *module-handle*—Returns **1** if *module-handle* is a valid modulespace in the current context; returns **0** otherwise.
- **modulespace inscope** *module-handle arg arg ...*—Executes a script in the context of a particular modulespace. This command is not expected to be used directly. Calls to it are generated implicitly when applications use the **modulespace code** command to create callback scripts to provide as context submodules.

The **modulespace inscope** command is much like the **modulespace eval** command except that it has *lappend* semantics and the modulespace must already exist. It treats the first argument as a list and appends any arguments after the first argument onto the end as proper list elements. A **modulespace inscope module-handle a x y z** command is equivalent to **modulespace eval module-handle [concat a [list x y z]]**. This *lappend* semantic is important because many calback scripts are actually prefixes.

- **modulespace parent**—Returns the module handle of the parent modulespace of the current modulespace.

Return Values

None

Command Completion

Immediate

Examples

```
modulespace new leg_incoming
modulespace terminate leg_incoming
modulespace listen leg_incoming
modulespace unlisten leg_incoming
modulespace return final PARAMgood|PARAMnull
modulespace return interim an-event PARAMgood|PARAMnull
modulespace event intercept leg_incoming
modulespace event consume leg_incoming
modulespace children $modHandle

set script [modulespace code {foo bar}]
modulespace code $script

modulespace current
modulespace eval module-x {foo bar x y}
modulespace exists $modHandle
modulespace inscope $modHandle $modScript
modulespace parent
```

Usage Notes

- None

object create dial-peer

Creates a list of dial-peer handles using **<***peer_handle_spec***>** as the prefix of the handle name.

Syntax

object create dial-peer <*peer_handle_spec*> <*destination_number*>

Arguments

- *peer_handle_spec*—Specifies the name of Tcl variables created to represent dial peer handles. The format of peer_handle_spec is <*handle_prefix*>*:*<*from_index*>. The system concatenates the prefix with a sequence number, starting with <*from_index*>, to build the dial peer handle name.
- *destination_number*—The call destination number.

Return Values

Returns the number of dial peer handles created.

Command Completion

Immediate.

Examples

```
object create dial-peer dp_handle:0 $dest
```

Usage Notes

- As an example of how the system generates handle names, consider the situation where two dial peers match the same destination. In this case, the return value will be 2, and the created handle names will be *dp_handle0* and *dp_handle1*.
- If a handle with a specified name already exists, the handle is deleted, regardless of its type, and a new handle is created.

object create gtd

Used to create a GTD Handle to a new GTD area from scratch. The system creates the associated underlying data structure ready for the application to insert (append) GTD parameters to it.

Syntax

object create gtd <*GTDHandle*> *{*<*message-id*>|<*reference-handle*>*}*

Arguments

- *GTDHandle*—The name of the handle the application wants to create and use for subsequent manipulations of the GTD message.
- *message-id*—The name of the message the application wants to create. The following values are supported:
  - IAM
  - CPG
  - ACM
  - ANM
  - REL
  - INF
  - INR
- *reference-handle*—Refers to an existing GTD handle; the format is: *&*<*handle_name*>.

Return Values

Returns the number; 1 if the handle can be created, 0 otherwise.

Command Completion

Immediate.

Examples

```
set gtd_creation_cnt [object create gtd gtd_setup_ind IAM]
set gtd_creation_cnt [object create gtd gtd_setup_ind2 &gtd_setup_ind]
```

Usage Notes

- This option is used if the application wants to build a GTD area from scratch. After creating the handle, the application typically appends one or more GTD attributes to it.
- The handle name must not contain the ':' character, because it has special meaning in the **object destroy** command.
- If a handle with the specified name already exists, it will be deleted (regardless of its type) before a new handle is created.
- As always, the application should check the return value before using the handle.
- A gtd handle cannot be handed off to another application.

object destroy

Destroys a specific dial peer item associated with *handle* or all handles specified by the *handle_spec*.

Syntax

object destroy [<*handle*> | <*handle_spec*>]

Arguments

- *handle*—The handle of the dial peer to be destroyed.
- *handle_spec*—Specifies a range of dial peer handles to delete. The format of *handle_spec* is <*handle_prefix*>:<*from_index*>:<*to_index*>. The system concatenates the prefix with the index and uses the result to delete the handle.

Return Values

Returns the number of objects destroyed.

Command Completion

Immediate.

Examples

```
object destroy dp_handle2
object destroy dp_handle:0:2
```

In the second example above, the system attempts to destroy dp_handle0, dp_handle1, and dp_handle2.

Usage Notes

- When a dial peer item, or a set of dial peers, is destroyed, the associated dial peer data is also destroyed.

object append gtd

Appends one or more GTD attributes to a handle.

Syntax

object append gtd <*GTDHandle*> <*GTDSpec*>

Arguments

- *GTDHandle*—the handle to the GTD area the application applies the modification to. The <GTDHandle> could be a handle that was created and assigned in a previous **infotag get evt_gtd** or could be one created from scratch using the **object create gtd** command.
- *GTDSpec*—the GTD attribute to modify.

Return Values

None

Command Completion

Immediate

Examples

```
object append gtd gtdhandleA &gtdhandleB.pci.-1
object append gtd gtdhandleA &gtdhandleB.pci.2
object append gtd gtdhandleA pci.1.dat "F4021234 " &gtdhandleB.fdc.-1
object append gtd gtdhandleA &gtdhandleB.fdc.-1 pci.1.dat "F4021234 "
```

Usage Notes

- When appending a GTD attribute instance to a GTD message, all fields of the GTD structure must be specified.
- As many attributes may be specified in a single gtd modification as the application wishes that does not exceed the limit of the Tcl parser. Use the backslash-newline sequence to spread a long command across multiple lines.
- If an attribute field is specified multiple times in a command, the value of the last processed attribute field will be used.
- The append command can have <instance_ref> as a <gtd_spec>.
- The <attr_instance> of an <instance_ref> does not contain field name. That is, operations involving an <instance_ref> always refer to the whole attribute.
- If multiple operations are applied to an attribute the result of the last operation may override the previous result. This is like doing multiple commands one after another.
- Any errors found during the syntax checking will abort the command.

object delete gtd

Deletes one or more GTD attributes.

Syntax

object delete gtd <GTDHandle> <GTD spec>

Arguments

- *GTDHandle*—the handle to the GTD area the application applies the modification to. The <GTDHandle> could be a handle that was created and assigned in a previous **infotag get evt_gtd** or could be one created from scratch using the **object create gtd** command.

- *GTDSpec*—the GTD attribute to modify.

Return Values

None

Command Completion

Immediate

Examples

```
object delete gtd gtdhandleA pci.1
object delete gtd gtdhandleA pci.-1
```

Usage Notes

- As many attributes can be specified in a single gtd modification as the application wants, as long as the limit of the Tcl parser is not exceeded. Use the backslash-newline sequence to spread a long command across multiple lines.
- If an attribute field is specified multiple times in a command, the value of the last processed attribute field will be used.
- The <attr_instance> in a delete command cannot specify a field name.
- The delete command does not accept <attr_value>.
- The delete command does not use <instance_ref> as <attribute_spec>.
- If multiple operations are applied to an attribute, the last operation overrides the previous result.
- Any errors found during syntax checking aborts this command.
- Deleting using the multiple instance form (-1) will not cause a script failure if no instance is found to delete. This allows scripts to works smoothly and quickly without checking for the existence of an attribute before deleting it.

object replace gtd

Replaces one or more GTD attributes.

Syntax

object replace gtd <GTDHandle> <GTD spec>

Arguments

- *GTDHandle*—The handle to the GTD area the application applies the modification to. The <GTDHandle> could be a handle that was created and assigned in a previous **infotag get evt_gtd** or could be one created from scratch using the **object create gtd** command.
- *GTDSpec*—the GTD attribute to modify.

Return Values

None

Command Completion

Immediate

Examples

```
object replace gtd gtdhandleA pci.1 &gtdhandleB.pci.5
object replace gtd gtdhandleA pci.-1 &gtdhandleB.pci.-1
object replace gtd gtdhandleA pci.-1 &gtdhandleB.pci.3
object replace gtd gtdhandleA pci.1 &gtdhandleB.pci.5    fdc.1.dat F4021234
object replace gtd gtdhandleA fdc.1.dat " F4021234" pci.1 &gtdhandleB.pci.5
```

Usage Notes

- As many attributes can be specified in a single gtd modification as the application wants, as long as the limit of the Tcl parser is not exceeded. Use the backslash-newline sequence to spread a long command across multiple lines.
- If an attribute field is specified multiple times in a command, the value of the last processed attribute field will be used.
- The <attr_instance> of an <instance_ref> does not contain field name. That is, operations involving an <instance_ref> always refer to the whole attribute.
- If multiple operations are applied to an attribute the result of the last operation may override the previous result. This is like doing multiple commands one after another.
- Any errors found during the syntax checking will abort the command.
- If <instance_ref> immediately follows an <attr_instance>, its value is used to update the specified <attr_instance>.
- If a reference handle is used, the script will not get a script error if the reference handle uses –1 as the instance number.

object get gtd

Retrieves the value of an attribute instance or a list of attributes associated with the specified GTD handle.

Syntax

object get gtd <*GTDHandle*> <*attr_instance*>

Arguments

- *GTDHandle*—The handle to the GTD area the application applies the modification to. The <GTDHandle> could be a handle that was created and assigned in a previous **infotag get evt_gtd** or could be one created from scratch using the **object create gtd** command.
- *attr_instance*—An attribute instance in the format: <*attr_name*>,<*field_instance*>,<*field_name*>.

Return Values

None

Command Completion

Immediate

Examples

```
object get gtd setup_gtd_handle pci.1.dat
object get gtd setup_gtd_handle fdc.-1.dat
```

Usage Notes

- If the application wants to retrieve the value of all instances of an attribute's field, it sets the content of <*field_instance*> to -1. If more than one instance is available, their values are separated by a space. Note that it does not matter if an attribute has multiple instances or not, a -1 will always be interpreted as "retrieve all instances."

object get dial-peer

Returns dial peer information of a dial peer item or a set of dial peers.

Syntax

object get dial-peer { <*handle*> | <*handle_spec*> } <*attribute_name*>

Arguments

- *handle*—The handle to the dial peer whose data is to be retrieved.
- *handle_spec*—Specifies a range of dial peer handles that and is of the format <*handle_prefix*>:<*from_index*>:<*to_index*>. Use this format to retrieve attribute information from a range of dial peer handles.
- *attribute_name*—Values can be one of the following:
  - encapType
  - voicePeerTag
  - matchTarget
  - matchDigitsE164
  - sessionProtocol

Return Values

A string containing the requested dial peer information. Depending on the command argument, either information about a set of dial peer handles or a specific one is returned. If information from more than one dial peer handle is returned, the values are separated by space.

Command Completion

Immediate.

Examples

```
object get dial-peer dp_handle3 matchTarget
object get dial-peer dp_handle:0:2 matchTarget
```

Usage Notes

- If the specified dial peer item does not exist or contain any dial peer, nothing is returned.
- The values for *encapType* can be one of the following:
  - Telephony
  - VoIP
  - Other (none of the above)
- The value for *voicePeerTag* is a number representing the peer item.

- The value for *matchTarget* is a string containing the configured target specification. For example, the value of matchTarget for a RAS session target is *session target ras*.
- The value for *matchDigitsE164* is a number string that matches the dial peer.
- The value for *sessionProtocol* can be one of the following:
  - H323
  - SIP
  - Other (none of the above)

param read

The **param read** command reads configuration parameters associated with the call into a variable with the name <*variable-name*>, which becomes read-only.

Syntax

param read <*variable-name*> [<package name>]

Arguments

- *package name*—Specifies the name of the package that executes the **package provide** command. If the package name is not specified, it implies that this command has been executed by a service.

Return Values

None

Command Completion

Immediate

Examples

```
param read userid
```

Usage Notes

None

param register

The **param register** command registers a parameter, with description and default values, allowing them to be configured and validated through the CLI. These commands are executed when the service or package is configured and loaded with commands such as **package provide**, which registers the capability of the configured module and its associated scripts.

Configured modules and their scripts are loaded and executed in slave interpreters to recognize and remember the packages they provide so they can be used when another service or package refers to this package. The **param register** command is also executed to recognize the parameters that the module registers to support.

Syntax

param register*<param-name>* [<param-description>] [<param-default>] [<param-type>]

Arguments

- *param-name*—The name of the parameter being registered.
- *param-description*—Parameter description.
- *param-type*—Currently restricted to three reserved types: string, integer, boolean. The syntax for specifying the type is: “s” | “i” | “b”, where “s”, “i”, or “b” designates the type of “string,” “integer,” or “boolean” correspondingly.
- *param-default*—Default value of the parameter.

Return Values

None

Command Completion

Immediate

Examples

```
param register uid-len “The user ID length” “7” “i”
```

Usage Notes

None

phone assign

The **phone assign** command binds the MAC address from the caller’s phone to a preexisting ephone template. This command is used with the extension assigner feature.

Syntax

phone assign {*legID* | *info-tag*} *tag*

Arguments

- *legID*—The ID of the call leg from which the MAC address will be retrieved and assigned to an ephone tag.
- *info-tag*—A direct mapped info-tag mapping to one leg.
- *tag*—ephone tag.

Return Values

1—Assignment succeeded.

2—Assignment failed.

Command Completion

Immediate

Example

```
set result [phone assign leg_incoming 20]
if {$result = "2"} puts "Assignment of 20 failed.\n"
```

Usage Notes

This command takes only one leg.

phone query

The **phone query** command verifies whether the ephone tag has been assigned a MAC address yet. This command is used with the extension assigner feature.

Syntax

phone query {*legID* | *info-tag*} ***-t*** *tag*

Arguments

- *legID*—The ID of the incoming call leg. This is used to identify the current caller/phone, so detailed assignment return values can be provided.
- *info-tag*—A direct mapped info-tag mapping to one leg.
- **-t** *tag*—ephone tag.

Return Values

- -1—Failed.
- 0—Invalid tag number.
- 1—Unassigned.
- 2—Assigned to the calling phone.
- 3—Assigned to other phone and phone is unregistered.
- 4—Assigned to other phone and phone is in idle state.
- 5—Assigned to other phone and phone is in use.

Command Completion

Immediate

Example

```
set result [phone query leg_incoming -t 20]
if {$result = "1"} puts "ephone 20 is available.\n"
```

Usage Notes

This command takes only one leg.

phone unassign

The **phone unassign** command removes the MAC address from the ephone tag. This command is used with the extension assigner feature.

Syntax

phone unassign {*legID* | *info-tag*} *tag*

Arguments

- *legID*—The ID of the call leg.
- *info-tag*—A direct mapped info-tag mapping to one leg.
- *tag*—ephone tag.

Return Values

1—Unassignment succeeded.

2—Unassignment failed.

Command Completion

Immediate

Example

```
set result [phone unassign leg_incoming 20]
if {$result = "2"} puts "Unassignment of ephone 20 failed.\n"
```

Usage Notes

This command takes only one leg.

playtone

The **playtone** command plays a tone on the specified call leg. If a conference is in session, the digital signaling processor (DSP) stops sending data to the remote end while playing a tone. This command is typically used to give the caller a dial tone if the script needs to collect digits.

Syntax

playtone {*legID* | *info-tag*} {*Tone* | *StatusCode*}

Arguments

- *legID*—The ID of the call leg to be handed off.
- *info-tag*— A direct mapped info-tag mapping to one or more legs. For more information about info-tags, see Chapter 4, "Information Tags."
- *Tone*—One of the following:
  - tn_none—Stops the tone that is currently playing.
  - tn_dial—Plays a dial tone.
  - tn_busy—Plays a busy tone.
  - tn_addrack—Plays an address acknowledgement tone.
  - tn_disconnect—Plays a disconnect tone.
  - tn_oos—Plays an out-of-service tone.
  - tn_offhooknotice—Plays an off-the-hook notice tone.
  - tn_offhookalert—Plays an off-the-hook alert tone.
- *StatusCode*—The status code returned by the evt_status info-tag. If a status code is specified, the **playtone** command plays the tone associated with that status code.

Return Values

None

Command Completion

Immediate

Example

```
playtone leg_incoming [getInfo evt_status]
playtone leg_all tn_oos
```

Usage Notes

- If the specified call leg is invalid, the script terminates and displays an error to the console, and the call is cleared.
- The **playtone** command only works for telephony call legs and is silently ignored for VoIP legs.

puts

The **puts** command outputs a debug string to the console if the IVR state debug flag is set (using the **debug voip ivr script** command).

Syntax

puts *string*

Arguments

- *string*—The string to output.

Return Values

None

Command Completion

None

Example:

```
puts "Hello $name"
```

requiredversion

The **requiredversion** command verifies that the script is running the correct version of the Tcl IVR API.

Syntax

requiredversion *majorversion.minorversion*

Arguments

- *majorversion*—Indicates the major version of the Tcl IVR API that the underlying Cisco IOS code supports.
- *minorversion*—Indicates the minimum level of minor version of the Tcl IVR API that the underlying Cisco IOS code supports.

Return Values

None

Command Completion

None

Example

```
requiredversion 2.5
```

Usage Notes

If the version of the script does not match the major version specified or is not equal to or greater than the minor version specified, the script terminates and an error is displayed at the console.

sendmsg

Sends a message to another application instance.

Syntax

sendmsg *{<app-name> | <handle>}* ***-p*** *<parameter_array>*

Arguments

- *<app-name>*—Creates a new instance using this application name.
- *<handle>*—The handle of an existing application instance.
- **-p** *<parameter_array>*—A Tcl array containing the list of parameters to pass.

Return Values

Returns “unavailable” or “success.”

Command Completion

Immediate.

Examples

```
set iid newapp
set fruit_message(text) "Request for Fruit"
set fruit_message(fruit) "Bananas"
set rval [sendmsg $iid -p fruit_message]
if $rval == "unavailable" {
    call close}
```

Usage Notes

- If the instance is not running on the gateway, it returns an “unavailable” return value.
- If an application name is provided, a new instance of that application is generated. The new instance will not have any active legs, but will receive an ev_msg_indication event.
- If the message is expected to generate a new instance of an application, but the gateway resources are not configured to allow new instances, the sendmsg command fails and clears all call legs it is handling. See the call treatment and call threshold commands in the Call Admission Control (CAC) document.

- The instance receiving the message, whether generated or already running, receives an ev_msg_indication event. The instance can then use the ev_msg and ev_msg_source information tags to retrieve more information.
- Messages cannot be sent to other gateways or servers.

service

Registers or unregisters a service.

Syntax

service *{register | unregister}* <*service-name*>

Arguments

- <*service-name*>—Name of the service.

Return Values

service register <*service-name*> returns "service already registered" or "registered."

service unregister <*service-name*> returns "service not registered" or "unregistered."

If a session tried to register or unregister a service name registered by another session, it receives the return value "service registered by another session."

Command Completion

Immediate.

Examples

```
set ret [service register cisco]
if {$ret="registered"} puts "Service successfully registered"

set ret [service unregister cisco]
if {$ret="unregistered"} puts "Service successfully unregistered"
```

Usage Notes

- This command puts the currently running handle into the service table.
- A second call to register the same service returns "service already registered."
- If the session terminates, the service is unregistered.
- A single session can register with multiple service-names. A second session registering with the same service-name returns "service already registered."
- A successful registration returns "registered."
- A list of registered services can be viewed by using the **show call application** CLI command.
- A Tcl script can find registered services using the *mod_handle_services* infotag.

set avsend

Sets an associative array containing standard AV or VSA pairs.

Syntax

set *avSend* (*attrName* [, *index*] **value**

Note Cisco IOS Release 12.1(2)T was the first release that incorporated the *avSend* argument.

Arguments

- *attrName*—Two IVR-specific attributes are supported: h323-ivr-out and h323-credit-amount. See the "AV-Pair Names" section on page 4-3 for more information on these types.
- *index*—An optional integer index starting from 0, used to distinguish multiple values for a single attribute.

Return Values

None

Command Completion

Immediate

Examples

```
set avsend(h323-credit-amount) 25.0

set avsend(h323-ivr-out,0) "payphone:true"
set avsend(h323-ivr-out,1) "creditTime:3400"
```

Usage Notes

If the specified call leg is invalid, the script terminates and displays an error to the console, and the call is cleared.

set callinfo

Sets the parameters in an array that determines how the call is placed. The outgoing call is then placed using the **leg setup** command.

Syntax

set callinfo *(tagName [,index]) value*

Arguments

- *tagName*—Parameter that determines how the call is placed. The array can contain the following:
  - *destinationNum*—Called or destination number. For **mode**, this argument is used as *transfer-target* or *forwarded-to* number. This parameter can accept a URL string. This parameter does not allow indexing.
  - *originationNum*—Origination number. For **mode**, this argument is used as *transfer-by* or *forwarded-by* number. This parameter can accept a URL string. This parameter does not allow indexing.

– *originationNumPI*—Calling number Presentation Indication value.

  Values allowed are:
  presentation_allowed
  presentation_restricted
  number_lost_due_to_interworking
  reserved_value

  This parameter does not allow indexing.

– *originationNumSI*—Calling number Screening Indication value.

  Values allowed are:
  usr_provided_unscreened
  usr_provided_screening_passed
  usr_provided_screening_failed
  network_provided

  This parameter does not allow indexing.

– *accountNum*—Caller's account number. This parameter does not allow indexing.

– *redirectNum*—Redirect number. Originally added to change a field in an end-to-end ISDN redirect IE. Also used to specify the number requesting a call transfer. Typically, the calling number of the leg that receives an **ev_transfer_request** event. Default value is *null*. This parameter does not allow indexing.

– *redirectNumPI*—Redirect number Presentation Indication value.

  Values allowed are:
  presentation_allowed
  presentation_restricted
  number_lost_due_to_interworking
  reserved_value

  This parameter does not allow indexing.

– *redirectNumSI*—Redirect number Screening Indication value.

  Values allowed are:
  usr_provided_unscreened
  usr_provided_screening_passed
  usr_provided_screening_failed
  network_provided

  This parameter does not allow indexing.

– *redirectCount<count>*—Used to set the redirect number Screening Indication value. Valid count values are in the range of 0–7. The count is automatically incremented with each forwarding request from the destination. The decision of when to stop forwarding at a specified count is the responsibility of the script. This parameter does not allow indexing.

– *redirectReason<value>*—Used to set the redirect number Reason value. This parameter does not allow indexing.

  Values allowed are:
  rr_no_reason
  rr_cfb
  rr_cfnr
  rr_rsvd1
  rr_rsvd2
  rr_rsvd3

rr_rsvd4
rr_rsvd5
rr_rsvd6
rr_rsvd7
rr_rsvd8
rr_rsvd9
rr_rsvd10
rr_ct
rr_cp
rr_not_present

In conjunction with **mode**, the following values specify the type while initiating call-forwarding:

rr_cfu
rr_cfb
rr_cfnr
rr_cd

- *redirectCfnrInd<value>*—Used to set the CFNR Indicator.

  Values allowed are:
  cfnr_true
  cfnr_false (default)

  This parameter does not allow indexing.

- *alertTime*—Determines how long (in seconds) the phone can ring before the call is aborted. The default is infinite. This parameter does not allow indexing.
- *usrDstAddr*—This tag maps directly to the destinationAddress in the user-to-user information of the H.323-Setup message. The tag can set this field in either e164 format or h323-id string format. A maximum of 10 instances of this tag is allowed. This parameter does not allow indexing.
- *usrSrcAddr*—This tag maps directly to the sourceAddress in the user-to-user information of the H.323-Setup message. The tag can set this field in either e164 format or h323-id string format. A maximum of 10 instances of this tag is allowed. This parameter does not allow indexing.
- *addrResSrcInfo*—This tag maps directly to *srcInfo* of the ARQ RAS message to the gatekeeper. The tag can set this field in either e164 format or h323-id string format. A maximum of 10 instances of this tag is allowed. This parameter does not allow indexing.
- *addrResDstInfo*—This tag maps directly to *dstInfo* of the ARQ RAS message to the gatekeeper. The tag can set this field in either e164 format or h323-id string format. A maximum of 10 instances of this tag is allowed. This parameter does not allow indexing.
- *displayInfo*—This tag maps directly to *displayInfo* of the H323-Setup message. This parameter does not allow indexing.
- *mode*—Possible values are: *rotary / redirect / redirect_rotary*. If not specified, the default value is *rotary*. This parameter does not allow indexing.
  - *rotary*—The call setup attempts to set up a call between the destination and the legID by normal call setup (rotary) routines and to conference the legs.
  - *redirect*—The call setup attempts to set up a call between the destination and the legID by transferring the legID endpoint to the destination phone number. A protocol-specific transfer request is sent on the legID to initiate the transfer. If the transfer attempt fails, the command aborts. It the transfer successful, the legID eventually gets disconnected from the endpoint, with the application relinquishing control of the leg as a side effect.

- *redirect_rotary*—The call setup attempts to set up a call between the destination and the legID by first transferring the legID endpoint to the destination phone number. If the transfer attempt fails, either internally by checking the type of call leg or after a transfer message round trip, the command tries to reach the destination by normal call setup (rotary) methods and to conference the legs. The application retains the control of the legID and the new leg. If the transfer is successful, the legID eventually gets disconnected from the endpoint, with the application relinquishing control of the leg as a side effect.

– *rerouteMode*—Possible values are: *none / rotary / redirect / redirect_rotary*. If not specified, the value is same as **mode**. If both this argument and **mode** are not specified, the default value is *rotary*. This parameter does not allow indexing.

- *none*—If the destination endpoint issues a redirect request while attempting a rotary call setup, the call setup aborts and an *ev_setup_done* event is sent to the script with redirected-to numbers. The redirect reason is specified in the *evt_redirect_info* information tag.
- *rotary*—If the destination endpoint issues a redirect request while attempting a rotary call setup, a normal rotary call setup occurs towards the redirected-to number.
- *redirect*—If the destination endpoint issues a direct request while attempting a rotary call setup, an attempt is made to propagate the request onto the legID. If the legID is not yet connected, a call-forwarding request is sent. If the legID is connected, a call-transfer request is sent. If the legID doesn't support any redirect mechanism, an *ev_setup_done* event with an appropriate error code is sent to the script.
- *redirect_rotary*—Similar to redirect, except that if the legID does not support any redirect mechanism, a normal rotary call setup occurs towards the redirected-to number.

– *transferConsultID*—A token used in call transfer with consultation. Typically extracted from an **ev_transfer_request** event. Default value is *null*. This parameter does not allow indexing.

– *notifyEvents*—A string of event names. Notify signaling messages listed in this parameter during rotary call setup and redirect call setup. Internally, call setup continues after reporting the event to the script. Default value is *null*. This parameter does not allow indexing.

– *originationNumTON*—Sets the calling number octet 3 TON field in the ccCallInfo structure. This parameter does not allow indexing.

Values allowed are:
ton_unknown
ton_international
ton_national
ton_network_specific
ton_subscriber
ton_reserved1
ton_abbreviated
ton_reserved2
ton_not_present

– *destinationNumTon*—Sets the called number octet 3 TON field in the ccCallInfo structure. This parameter does not allow indexing.

  Values allowed are:
  ton_unknown
  ton_international
  ton_national
  ton_network_specific
  ton_subscriber
  ton_reserved1
  ton_abbreviated
  ton_reserved2
  ton_not_present

– *originationNumNPI*—Sets the calling number octet 3 NPI field in the existing ccCallInfo structure. This parameter does not allow indexing.

  Values allowed are:
  npi_unknown
  npi_isdn_telephony_e164
  npi_reserved1
  npi_data_x121
  npi_telex_f69
  npi_reserved2
  npi_reserved3
  npi_reserved4
  npi_national_std
  npi_private
  npi_reserved5
  npi_reserved6
  npi_reserved7
  npi_reserved8
  npi_reserved9
  npi_reserved10
  npi_not_present

– *destinationNumNPI*—Sets the called number octet 3 NPI field in the existing ccCallInfo structure. This parameter does not allow indexing.

  Values allowed are:
  npi_unknown
  npi_isdn_telephony_e164
  npi_reserved1
  npi_data_x121
  npi_telex_f69
  npi_reserved2
  npi_reserved3
  npi_reserved4
  npi_national_std
  npi_private
  npi_reserved5
  npi_reserved6
  npi_reserved7
  npi_reserved8

npi_reserved9
npi_reserved10
npi_not_present

- *guid*—The GUID of the outgoing call leg. This parameter does not allow indexing.
- *incomingGuid*—The *incoming GUID* field for the outgoing call leg. This parameter does not allow indexing.
- *originalDest*—The original called number. This parameter does not allow indexing.
- *protoHeaders*—An array containing the header av-pair to be sent in the call setup.
- *newguid*—Set to *true* to specify that a new GUID should be generated and used for the outgoing call setup. By default, a new GUID is not generated for the outgoing call.

- *index*—An optional integer, starting with 0, used to distinguish multiple instances of a single tag.
- *value*—The value to be set.

Return Values

None

Command Completion

Immediate

Examples

```
set callInfo(usrDstAddr,0) "e164=488539663"
set callInfo(addrResSrcInf,1) "h323Id=09193926573"
set callInfo(displayInfo) "hi there"
set callInfo(mode) "REDIRECT_ROTARY"
set callInfo(rotaryRedirectMode) "ROTARY"
set callInfo(notifyEvents) "ev_transfer_status ev_alert"
set callInfo(transferConsultID) $targetConsultID

set setupSignal(Subject) "Hotel Reservation"
set setupSignal(Priority) "urgent"
set setupSignal(X-ReferenceNumber) "1234567890"
set callInfo(protoHeaders) setupSignal
```

Usage Notes

- The name *callInfo* is a convention in Tcl scripts for the **leg setup** command, but the name is not enforced by Cisco IOS software. The name can be different. For example:

  ```
  set dest "5550100"
  set myInfoForCallSetup(mode) "REDIRECT_ROTARY"
  leg setup $dest myInfoForCallSetup
  ```

- The Tcl **set** command does not perform any call setup argument checking, since the code does not start the call setup until the **leg setup** command is executed. For example:

  ```
  set callInfo(redirectCount) BadValue
  ```

  does not cause an error nor will it fail the call. The call fails when the **leg setup** command is thereafter executed.

subscription open

Sends a subscription request to a subscription server.

Syntax

subscription open *{URL} subscriptionInfoArray* ***-s*** *subscription_id*

Arguments

- *URL*—URL of the server to send the subscription request to. Only SIP URLs are supported.
- *subscriptionInfoArray*—An array containing attributes about the subscription. Can contain any of the following:
  - *event*—Name of event to be subscribed.
  - *expirationTime*—Time after the subscription expires, in seconds.
  - *protoHeaders*—An array containing headers to be sent in the subscription request.
  - *subscriptionContext*—An array containing av-pairs on the context of a subscription. This argument allows the subscribing system to specify a list of av-pairs as context information that can be useful to the module or application that receives the notification. The array can contain the following:
  - *content_type*—The type of content, such as plain or XML. Only textual content is supported.
  - *content*—A string that has significance only to the application. The content can be any information in the form of av-pairs or any other format specified by the content_type. The content is sent in the protocol message body. Only textual content is supported.
  - *notificationReceiver*—This argument takes either the *appName* or *moduleHandle* attribute. If the name of the application is specified and the application is configured, that application is generated to receive notification. The *moduleHandle* attribute specifies the running instance of a module or session. The moduleHandle can be obtained using the **infotag get mod_handle** command. This handle represents a running instance of an application.
- *-s subscription_id*—ID of the subscription. This argument takes the subscription ID as the parameter and is used for resubscription when the subscription already exists.

Return Values

subscriptionID—A unique ID assigned to this subscription.

Command Completion

When this command finishes, the script receives an ev_subscribe_done event.

Examples

```
set subcriptionInfoArray (notificationReceiver) notifyApp
set mySubID [subscription open sip:my_id@cisco.com subscriptionInfoArray]
```

The following example sends a subscribe request to the server for the event package "msg:"

```
-----------------------------------------------------------
set url sip:foo@xyz.com
# set the event-package
set subinfo(event) msg

#set the expiration time for the subscription in seconds
set subinfo(expirationTime) 500

# specify a header
```

```
set headers(Subject) "Hi There"
set subinfo(protoHeaders) headers

# specify the content
set subinfo(protoContentTypes) "text/plain"
set subinfo(protoContents) "This is from client"

#set context information for subscription
set context(actNum) 1234
set context(pinNum) 5678
set subinfo(subscriptionContext) context

# send the request
subscription open $url subinfo
-------------------------------------------------------------
```

Usage Notes

- Tcl IVR 2.0 limits the number of subscriptions per handler to 18. Because each script instance is a handler, an application instance can only handle a maximum of 18 subscriptions simultaneously.
- The user can specify how to handle the notification received from the server in one of the following ways:
  - The current script instance that is doing the subscription can handle the notification. For this to happen, do not specify either the application name (appName) or the moduleHandle in the arguments.
  - A new application instance, whether in the same application or in a different application, can be created to handle the notification. For this to happen, specify the application name (appName) in the arguments.
  - A different running application instance can handle the notification. For this to happen, specify the moduleHandle in the arguments.
- The application that makes the subscription is the controlling application. For example, it handles the notification and removes the subscription.
- To make another application take over control of the application, the application that made the subscription must close. For example, application A makes the subscription and specifies “notificationReceiver” to be application B. Unless application A closes by calling “call close,” the notification is not sent to application B. The same applies if A specifies a moduleHandle.
- A script can pass the legID associated with a leg to the subscription request being made. This allows debugging based on a leg.
- Event and expirationTime are mandatory arguments that the script must specify.
- Context information is not sent to the server, it is kept along with the subscription information. For example, information specific to a user, such as accountNumber or pinNumber, is kept within the subscription. Context information is deleted whenever the subscription is removed.

subscription close

Removes an existing subscription.

Syntax

subscription close *subscription_id*

Arguments

- *subscription_id*—ID of the subscription to close.

Return Values

None

Command Completion

When this command completes, the script receives an ev_unsubscribe_done event.

Examples

```
set mySubID openSub
subscription close mySubID
```

Usage Notes

None

subscription notify_ack

Sends a positive or negative acknowledgment for a notification event.

Syntax

subscription notify_ack <*subscription_id*> *[-i notifyAckInfo]*

Arguments

- *subscription_id*—ID of the subscription.
- *-i notifyAckInfo* - An associative array that can contain the following:
  - *protoHeaders*—Header information.
  - *protoContents*—Content information.
  - *protoContentTypes*—Content type information.
  - *respCode*—Valid values are *ack* or *nak*. If unspecified, the default value of *ack* is assumed. Ack sends a positive acknowledgment for notification and nak rejects the notification. When the application rejects the notification, it should insert headers, such as 'Warning,' so that the appropriate reason is sent to the server.

Return Values

None

Command Completion

Immediate

Examples

```
set mySubID [infotag get evt_subscription_id]
set headers(Hello) "Hello, this is ACK header"
set ackinfo(protoHeaders) headers
set ackinfo(respCode) "ack"
subscription notify_ack $mySubID -i ackinfo
```

Usage Notes

None

timer left

The **timer left** returns the number of seconds left on the timer associated with the name.

Syntax

timer left *type [name]*

Arguments

- *type*—The type of timer, such as *named_timer.*
- *name*—A string name associated with this timer as the key for association.

Return Values

Number of seconds left on the timer.

Command Completion

None

Examples

```
timer left named_timer timer_1
timer left named_timer 1
```

Usage Notes

None

timer start

The **timer start** command starts a timer for a specified number of seconds. Each timer is associated with a name as its key, allowing multiple named_timers for each script.

Syntax

timer start *type time [name]*

Arguments

- *type*—The type of timer, such as *named_timer.*
- *time*—The time, in seconds, that the timer should run.
- *name*—A string name associated with this timer as the key for association.

Return Values

None

Command Completion

When the timer expires, the script receives an ev_named_timer event. The name associated with this named_timer can be retrieved using the *evt_timer_name* information tag

Examples

```
timer start named_timer 60 timer_1
timer start named_timer 100 1
```

Usage Notes

- If another timer is still running, this command stops the previous timer and start the specified timer.

timer stop

The **timer stop** command stops the timer associated with the name.

Syntax

timer stop *type [name]*

Arguments

- *type*—The type of timer, such as *named_timer*.
- *name*—A string name associated with this timer as the key for association.

Return Values

None

Command Completion

None

Examples

```
timer stop named_timer timer_1
timer stop named_timer 1
```

Usage Notes

None

CHAPTER 4

Information Tags

Information tags (info-tags) are identifiers that can be used to retrieve information about call legs, events, the script itself, current configuration, and values returned from RADIUS.

Note Some info-tags have one or more parameters that are used to further identify the information to be retrieved, set, or modified.

Info-tags are grouped according to use. The first three characters of the info-tag label indicate the grouping:

- aaa—RADIUS information.
- cfg—Configuration information.
- con—Connection information.
- evt—Event information.
- leg—Call leg information.
- med—Media services information.
- sys—System information.

This chapter lists the available info-tags and the following information about each:

- Description—Explanation of the purpose of the info-tag.
- Syntax—The syntax of the info-tag.
- Mode—Whether the info-tag is read or read-write.
- Scope—The context in which the info-tag can be used. Some info-tags can be used at any time (global). Others are valid only when certain events are received, and the script terminates with error output if the info-tag is used in other situations. For example, you cannot call evt_dcdigits while handling the ev_setup_done event. In other words, if the previous command is **leg setup** and the ev_setup_done event has not yet returned, then you cannot execute an **infotag get evt_dcdigits** command, or the script will terminate with error output.
- Return Type—The type of information returned by the info-tag when used with an **infotag get** or **infotag set** command.
- Direct Mapping—Whether the info-tag can be used directly with a command (other than the **infotag get** or **infotag set** commands) and with which commands it can be used.

Note If an info-tag is specified incorrectly, if any of the parameters are specified incorrectly, or if the info-tag is used outside its intended scope, the script terminates with error output.

aaa_accounting_last_sent

Description	Retrieves the timestamp of the last accounting record sent from the voice-aaa subsystem.
Syntax	**infotag get aaa_accounting_last_sent** *{servertag}* • *servertag*—The server or server group identifier. This value refers to the method-list name, as in the following AAA configuration: `aaa accounting connection {default | method-list-name} group group-name`
Mode	Read
Scope	Valid only on completion of a successful servertag subscription.
Return Type	timestamp
Direct Mapping	None

aaa_avpair

Description	Returns the value of an AV-pair that was returned by RADIUS. After an **authorize** command finishes, the RADIUS server could have returned parameters as AV-pairs. This info-tag, along with aaa_avpair_exists, is used to get the value of a parameter after checking that such a parameter was returned. Refer to the table in "AV-Pair Names" section on page 4-3 for a list of valid VSA AV-pair names.
Syntax	**infotag get aaa_avpair** *avpair-name*
Mode	Read
Scope	Global
Return Type	String, Number, Boolean (1 or 0), or any other value that is configured or returned through RADIUS.
Direct Mapping	None

aaa_avpair_exists

Description	Returns the number of matched AV-pairs in the RADIUS server return. After an **authorize** command completes, the RADIUS server may return parameters as AV-pairs. This info-tag, along with aaa_avpair, is used to find out if a parameter exists before getting its value. Refer to the table in the "AV-Pair Names" section on page 4-3 for a list of valid VSA AV-pair names.
Syntax	**infotag get aaa_avpair_exists** *avpair-name*
Mode	Read
Scope	Global
Return Type	Number
Direct Mapping	None

AV-Pair Names

The info-tag aaa_avpair_exists can be used to check the availability of a VSA. The info-tag aaa_avpair can be used to access the value returned in this VSA. The valid VSA names that can be passed as parameters to these commands are the following.

Type	Name	Description
aaa	h323-ivr-in	A generic VSA for the billing server to send any information to the gateway in the form of an AV-pair, such as "color:blue" or "advprompt:rtsp://www.cisco.com/rtsp/areyouready.au"
	h323-ivr-out	A generic VSA for the gateway to send any information to the billing server in the form of an AV-pair, such as "color:blue" or "advprompt:rtsp://www.cisco.com/rtsp/areyouready.au"
	h323-credit-amount	The credit amount remaining in the account is returned.
	h323-credit-time	The credit time remaining in the account is returned.
	h323-prompt-id	The ID of the prompt is returned.
	h323-redirect-number	The number for redirection of a call is returned.
	h323-redirect-ip-addr	The IP address for the preferred route is returned.
	h323-preferred-lang	The language that the billing system returns as the preferred language of the end user. Three languages are supported; en (english), sp (spanish), and ch (mandarin). You can define additional languages as needed.
	h323-time-and-day	The time and day at the destination.
	h323-return-code	This information is returned only after an **authorization** command is issued. It returns either a numerical value or "Unknown variable name." The numerical value indicates what action the IVR application should take, namely to play a specific audio file to inform the end user of the reason for the failed authorization. If "Unknown variable name" is returned, the external AAA-server is out of service.
	h323-billing-model	Indicates the billing model used for the call. Initial values: 0=Credit, 1=Debit. Note: The debit card application assumes a Debit billing model.
	h323-currency	ISO currency to indicate what units to use in playing the remaining balance. The debit card application assumes units of *preferred_language*_dollar.au and *preferred_language*_cent.au.

Note If the *aaa* variable returns "0," this indicates that there is no VSA match to the name returned.

aaa_new_guid

Description	Request the system to generate and return a new GUID.
Syntax	**infotag get aaa_new_guid**
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

cfg_avpair

Description	Returns the value of an AV-pair that was configured through the CLI.
Syntax	**infotag get cfg_avpair** *avpair-name*
Mode	Read
Scope	Global
Return Type	String, Number, Boolean (1 or 0), or any other value that is configured or returned through RADIUS.
Direct Mapping	None

cfg_avpair_exists

Description	Returns an indication of whether the specified parameter or AV-pair was configured through the CLI.
Syntax	**infotag get cfg_avpair_exists** *avpair-name*
Mode	Read
Scope	Global
Return Type	Boolean (1 = true; 0=false)
Direct Mapping	None

con_all

Description	Returns or maps to a list of all the connection IDs in the script.
Syntax	**infotag get con_all**
Mode	Read
Scope	Global
Return Type	Number list
Direct Mapping	Connections

con_ofleg

Description	Gets a list of all the connections the leg is a part of. This does not include those connections that are in Creation or under Destruction. The info-tag should map to just one leg.
Syntax	**infotag get con_ofleg** {*info-tag* | *legID*}
Mode	Read
Scope	Global
Return Type	Number list
Direct Mapping	Connections

evt_aaa_status_info

Description	Retrieves the aaa information. For example, the method list from the *ev_accounting_status_ind* event.
Syntax	**infotag get evt_aaa_status_info** *[attribute-name]* • *attribute-name*—The attribute you want to access. Defined attributes are *method-list-name* and *status*. If no attribute is specified, the list of attributes returns in the form of *av1#av2#*, where "#" is the delimiter. Status values are: – 000—Accounting Failed – 001—Accounting Success
Mode	Read
Scope	ev_accounting_status_ind
Return Type	String
Direct Mapping	None
Examples	set method_list [infotag get evt_aaa_status_info method-list-name]

evt_address_resolve_reject_reason

Description	Returns the address resolution rejection cause.
Syntax	**infotag get evt_address_resolve_reject_reason**
Mode	Read
Scope	ev_address_resolved
Return Type	Number
Direct Mapping	None

evt_address_resolve_term_cause

Description	Returns the address resolution termination cause.
Syntax	**infotag get evt_address_resolve_term_cause**
Mode	Read
Scope	ev_address_resolved
Return Type	Number
Direct Mapping	None

evt_connections

Description	Returns a list of connection IDs associated with the event received.
Syntax	**infotag get evt_connections**
Mode	Read
Scope	ev_handoff ev_returned ev_setup_done ev_create_done ev_destroy_done
Return Type	Number list
Direct Mapping	Connections

evt_consult_info

Description	Returns consult information from a consult response event.
Syntax	**infotag get evt_consult_info** {*consultID* | *transferDest*}
Mode	Read
Scope	ev_consult_response
Return Type	String
Direct Mapping	None

evt_dcdigits

Description	Returns the digits collected by the **leg collectdigits** command.
Syntax	**infotag get evt_dcdigits**
Mode	Read
Scope	ev_collectdigits_done
Return Type	String
Direct Mapping	None

evt_dest_handle

Description	Returns the application handle of the instance that registered for the destination number. This value is only available when the destination is an analog FXS phone and the phone is busy. If no application registered for the destination number, the value is an empty string.
Syntax	**infotag get evt_dest_handle**
Mode	Read
Scope	ev_setup_done
Return Type	String; represents a TCL application handle
Direct Mapping	None

evt_digit

Description	Returns the digit key that was pressed.
Syntax	**infotag get evt_digit**
Mode	Read
Scope	ev_digit_end
Return Type	String
Direct Mapping	None

evt_digit_duration

Description	Returns the duration of the digit that was pressed.
Syntax	**infotag get evt_digit_duration**
Mode	Read
Scope	ev_digit_end
Return Type	Number
Direct Mapping	None

evt_disc_iec

Description	Returns the Internal Error Code (IEC). In the case of multiple IECs, only the last one is returned. The IEC returns in the same format as Radius VSAs, as a dotted "version.entity.category.subsystem.error.diagnostic" string. If <*component*> is not specified, this command returns the dotted string with the components of the IEC. For example, "1.1.180.3.21.0." If there is no internal error associated with the disconnect, the returned string is "0.0.0.0.0.0."
Syntax	**infotag get evt_disc_iec** [<*component*>] The optional <*component*> argument can be one of the following parameters: • *entity*—Returns the entity component of the IEC. For example, "1." • *category*—Returns the category component of the IEC. For example, "180." • *subsystem—R*eturns the subsystem component. For example, "3." • *errcode*—Returns the subsystem-specific error code component. For example, "21."
Mode	Read
Scope	ev_disconnected
Return Type	String
Direct Mapping	None

evt_disc_rsi

Description	Returns the RSI numeric value that indicates where the release originated from. If there is no RSI available, this command returns 0. **Note** Valid only upon receiving an *ev_disconnected* event. If called while processing any other event, this command returns a TCL_ERROR, causing the script to terminate.
Syntax	**infotag get evt_disc_rsi**
Mode	Read
Scope	ev_disconnected
Return Type	String
Direct Mapping	None

evt_endpoint_addresses

Description	Returns a list of endpoint addresses.
Syntax	**infotag get evt_endpoint_addresses**
Mode	Read
Scope	ev_address_resolved
Return Type	String The return value has the following structure: <*endpointAddress*>#<*endpointAddress*>#... The first *endpointAddress* is the primary address. The *endpointAddresses* that follow are the alternate addresses.
Direct Mapping	None

evt_event

Description	Returns the name of the event received.
Syntax	**infotag get evt_event**
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

evt_facility_id

Description	Returns the service type of the facility message response. The value is *ss_mcid_resp* for MCID invocation responses.
Syntax	**infotag get evt_facility_id**
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None
Example	set facility_id [infotag get evt_facility_id]
Usage Notes	None

evt_facility_report

Description	Enables the receipt of facility events.
Syntax	**infotag set evt_facility_report <***mcid* | *gtd***>**
Mode	Write
Scope	Global
Return Type	String
Direct Mapping	None
Example	infotag set evt_facility_report gtd
Usage Notes	• The *mcid* option of this information tag must be set to receive facility responses from MCID responses. • The *gtd* option of this information tag must be set to receive facility events that contain GTD information.

evt_feature_param

Description	Returns parameters related to a specific feature event.
Syntax	**infotag get evt_feature_param** *{parameter_name}* Possible return strings for media_inactivity are as follows: *no media received*—Media inactivity detected; no RTP or RTCP packets have been received for a configured amount of time. RTCP packet has been received before a media inactivity condition is met. *no control info received*—Media inactivity detected; no RTP or RTCP packets have been received for a configured amount of time. No RTCP packet has been received before a media inactivity condition is met.
Mode	Read
Scope	ev_feature
Return Type	String
Direct Mapping	None
Example	infotag get evt_feature_param media_inactivity_type

evt_feature_report

Description	Enables or disables certain feature events to be intercepted by the script.
Syntax	**infotag set evt_feature_report** *{["no_"]event_names}* • event_names—A list of application event names that define what events should or should not be reported to an application when a call is active. An event name with a *"no_"* prefix means not to report it. Possible values for event_names are as follows: – fax – modem – modem_phase – hookflash – onhook – offhook – media_inactivity
Mode	Write
Scope	ev_feature
Return Type	None
Direct Mapping	None
Examples	To enable hookflash and disable fax and modem feature events received by the script: `infotag set evt_feature_report hookflash nofax nomodem` To enable media_inactivity received by the script: `infotag set evt_feature_report media_activity`

evt_feature_type

Description	Returns the feature type string when a feature event is received.
Syntax	**infotag get evt_feature_type** • Possible event names returned are as follows: – fax – modem – modem_phase – hookflash – onhook – offhook – media_inactivity – media_activity
Mode	Read
Scope	ev_feature
Return Type	String
Direct Mapping	None

evt_gtd

Description	Associates a handle to the GTD parameters contained in the event. The application can use the handle to include the associated GTD parameters in any outgoing call signal message.
Syntax	**infotag get evt_gtd <gtd_handle>**
Mode	Read
Scope	ev_address_resolved ev_alert ev_connected ev_disconnected ev_proceeding ev_progress ev_setup_indication
Return Type	Number. If a handle can be created from the event, 1 is returned, otherwise 0 is returned.
Direct Mapping	None
Example	set handle [infotag get evt_gtd gtd_inf]
Usage Notes	None

evt_handoff ani

Description	Returns the ani set by the inbound application in the <transfer>/ field of the **leg setup** command.
Syntax	**infotag get evt_handoff ani**
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None
Example	`set ani [infotag get evt_handoff ani]`
Usage Notes	This command is only available in the handoff event.

evt_handoff argstring

Description	This command replaces the existing *evt_handoff_string* information tag.
Syntax	**infotag get evt_handoff argstring**
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None
Example	`set hstring [infotag get evt_handoff argstring]`
Usage Notes	This command is only available in the handoff event.

evt_handoff dnis

Description	Returns the dnis set by the inbound application in the <transfer>/ field of the **leg setup** command. Available only in the handoff event.
Syntax	**infotag get evt_handoff dnis**
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None
Example	`set dnis [infotag get evt_handoff dnis]`
Usage Notes	None

evt_handoff_legs

Description	Returns all the legs handed off to the application. Typically used to retrieve all call legs in a multiple-leg handoff, but can also be used for a single-leg handoff.
Syntax	**infotag get evt_handoff_legs**
Mode	Read
Scope	ev_handoff
Return Type	String; represents legIDs separated by space characters.
Direct Mapping	None

evt_handoff proto_headers

Description	Retrieves the handoff header.
Syntax	**infotag get evt_handoff proto_headers** [<*attribute-name*>] <*attribute-name*>—Name of the header to get.
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

Example	`set AccountInfo [infotag get evt_handoff proto_headers AccountInfo]`
	The following command returns all headers in a concatenated string. Each header av-pair is separated by a '&':
	`set allHeaders [infotag get evt_handoff proto_headers]`

Usage Notes	If <attribute-name> is not specified, all headers are returned in a concatenated string, with each header separated by a "&" symbol.

evt_handoff_string

Description	Returns the handoff string when one or more call legs are handed off or returned to the script.
Syntax	**infotag get evt_handoff_string**
Mode	Read
Scope	ev_handoff ev_returned
Return Type	String
Direct Mapping	None

evt_iscommand_done

Description	Returns an indication of whether the command has finished.
Syntax	**infotag get evt_iscommand_done**
Mode	Read
Scope	ev_returned ev_setup_done ev_collectdigits_done ev_vxmldialog_done
Return Type	Boolean (1 = true; 0 = false)
Direct Mapping	None

evt_last_disconnect_cause

Description	Returns the value of the last failure detected during this call. The failure could have occurred on any call leg associated with this call. If no failures have occurred during the call, *di_000* is returned. The value of this information tag is updated while processing the following events: • ev_disconnected—Set to the cause value recieved in the protocol message. • ev_disc_prog_ind—Set to the cause value recieved in the protocol message. • ev_collectdigits_done—Set to *di_028* (invalid number) when the ev_collectdigits_done event returns status *cd_006*. Not modified when other digit collect status codes are returned. • ev_setup_done—Set to the cause code associated with the call setup attempt. The value is *di_016* (normal) if the call setup is successful. • ev_authenticate_done—Set to *di_057* (bearer capability is not available) when the ev_authenticate_done event status is not *au_000*. Not modified if event status is *au_000*. • ev_authorize_done—Set to *di_057* (bearer capability is not available) when the ev_authorize_done event status is not *ao_000*. Not modified if event status is *ao_000*.
Syntax	**infotag get evt_last_disconnect_cause**
Mode	Read
Scope	Global
Return Type	String. See the "Disconnect Cause" section on page 5-8 for string format.
Direct Mapping	None

evt_last_event_handle

Description	Returns the command handle of the setup.
Syntax	**infotag get evt_last_event_handle**
Mode	Read
Scope	ev_address_resolved ev_alert
Return Type	String
Direct Mapping	None

evt_last_iec

Description	When the script receives an *ev_setup_done* event that returns a bad status and wants more information, the script can poll this infotag to find the last Internal Error Code (IEC) associated with the rotary setup attempts and play an appropriate message to the caller on the incoming leg based on the IEC. This infotag is valid for any event. If there is no IEC associated with the call yet, the returned string is 0.0.0.0.0.0, or 0 for component.
Syntax	**infotag get evt_last_iec** [<*component*>] The optional <*component*> argument can be one of the following parameters: • *entity*—Returns the entity component of the IEC. For example, "1." • *category*—Returns the category component of the IEC. For example, "180." • *subsystem*—Returns the subsystem component. For example, "3." • *errcode*—Returns the subsystem-specific error code component. For example, "21."
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

evt_legs

Description	Returns a list of leg IDs associated with the event received. For information about which legs the evt_legs info-tag returns for a specific event, see Chapter 5, "Events."
Syntax	**infotag get evt_legs**
Mode	Read
Scope	ev_authorize_done ev_leg_timer ev_digit_end ev_hookflash ev_disconnected ev_disconnect_done ev_grab ev_setup_indication ev_media_done ev_handoff ev_returned ev_setup_done ev_collectdigits_done ev_vxml_dialog_done ev_vxmldialog_event
Return Type	Number list
Direct Mapping	Legs

evt_module_handle

Description	Returns the module handle of the module that generated the current module event.
Syntax	**infotag get evt_module_handle**
Mode	Read
Scope	ev_module_done, ev_module_event
Return Type	String
Direct Mapping	None

evt_module_subevent

Description	Returns the module subevent name specified in the **module return interim** command that generated the *ev_module_event* being processed. This provides access to the parameters associated with the specific module event being handled.
Syntax	**infotag get evt_module_subevent**
Mode	Read
Scope	ev_module_event
Return Type	String
Direct Mapping	None

evt_module_context

Description	Returns the module context specified when creating this module with the **module new** command.
Syntax	**infotag get evt_module_context**
Mode	Read
Scope	ev_module_done, ev_module_event
Return Type	String
Direct Mapping	None

evt_msg

Description	Retrieves the message body. **Note** Only valid when handling an ev_msg_indication event.
Syntax	**infotag get evt_msg** <*array-name*> <array_name>—The name of a TCL array populated with information from the -p attribute in the sendmsg command.
Mode	Read
Scope	Global
Return Type	A list of array item names.
Direct Mapping	None

Examples	`proc act_handle_msg {`
	`  infotag get **evt_msg** rx_message`
	`  if { [info exists rx_message(fruit)] == 1 } {`
	`  set fruit $rx_message(fruit)`
	`  } else { set fruit "peaches" }`
	`  set ident [infotag get evt_msg_source]`
	`  set return_msg(text) "Got your message requesting $fruit, ignoring it"`
	`  sendmsg $ident -p return_msg`
	`}`

Usage Notes	None

evt_msg_source

Description	Retrieves the handle of the source of the message.
Syntax	**infotag get evt_msg_source**
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

Examples	`proc act_handle_msg {`
	`  infotag get evt_msg rx_message`
	`  if { [info exists rx_message(fruit)] == 1 } {`
	`  set fruit $rx_message(fruit)`
	`  } else { set fruit "peaches" }`
	`  set ident [infotag get` **`evt_msg_source`**`]`
	`  set return_msg(text) "Got your message requesting $fruit, ignoring it"`
	`  sendmsg $ident -p return_msg`
	`}`

Usage Notes	• This handle is the same kind of handle returned by the *mod_handle* information tag. It can be used in a **sendmsg** or **handoff** command. • This information tag is only valid after an ev_msg_indication event. If called at other times, an error occurs and the script fails.

evt_params

Description	Creates an array variable named <*array-name*> within the current modulespace from the information provided by the <*param-array*> parameter of the **module return** command. This provides access to the parameters associated with the specific module event being handled.
Syntax	**infotag get evt_params** <*array-name*>
Mode	Read
Scope	ev_module_done, ev_module_event
Return Type	String
Direct Mapping	None

evt_progress_indication

Description	Returns the value of the progress indication of the received alert, connected, disconnect, disconnect with PI, proceeding, or progress message.
Syntax	**infotag get evt_progress_indication**
Mode	Read

Scope	ev_alert ev_connected ev_progress ev_proceeding ev_disconnected ev_disc_prog_ind
Return Type	Number
Direct Mapping	None
Example	set progress [infotag get evt_progress_indication]
Usage Notes	None

evt_proto_content

Description	Used to retrieve the content of the received event. The content is the body of the protocol message.
Syntax	**infotag get evt_proto_content**
Mode	Read
Scope	ev_notify, ev_subscribe_done, ev_unsubscribe_indication
Return Type	String
Direct Mapping	None
Examples	`set r_content [infotag get evt_proto_content]`
Usage Notes	Only textual content, such as plain text or xml text, is supported.

evt_proto_content_type

Description	Used to retrieve the content type of the received event. This event specifies the type of content carried in the body of the protocol message.
Syntax	**infotag get evt_proto_content_type**
Mode	Read
Scope	ev_notify, ev_subscribe_done, ev_unsubscribe_indication
Return Type	String
Direct Mapping	None
Examples	`set content_type [infotag get evt_proto_content_type]`
Usage Notes	Only textual content, such as plain text or xml text, is supported.

evt_proto_headers

Description	Used to access the protocol header information associated with the events.
Syntax	**infotag get evt_proto_headers** <*attribute-name*> <*attribute-name*>—Name of the header to get.
Mode	Read
Scope	ev_notify, ev_subscribe_done, ev_unsubscribe_indication
Return Type	String containing the value of the header attribute.
Direct Mapping	None

Examples	`set event [infotag get evt_proto_headers Event]`
	`set inviteSubject [infotag get evt_proto_headers "Subject"]`
	`set all_headers [infotag get evt_proto_headers]`

Usage Notes	• Both standard and non-standard headers can be accessed using this command. The application does not cache the header values. If an application wants to retain header information, it must save the information in its local or global variables. • If <attribute-name> is not specified, all headers are returned in a concatenated string, with each header separated by a "&" symbol.

evt_report ev_transfer_request

Description	Allows notification of the call transfer request event from an endpoint to the application.
Syntax	**infotag set evt_report ev_transfer_request**
Mode	Write
Scope	Global
Return Type	String
Direct Mapping	None
Examples	`infotag set evt_report ev_transfer_request`
Usage Notes	• The script performs a leg setup to the transfer-to endpoint. The callinfo (transferCall, transferBy, consultID) field is populated with information available in the evt_transfer_info information tag. • After the callinfo field is populated, a call is set up toward the transfer-to endpoint.

evt_redirect_info

Description	Returns forwarding request information when a call is being forwarded.
Syntax	**infotag get evt_redirect_info** {*redirectDest* | *redirectReason* | *redirectCount* | *originalDest*} • redirectDest—redirected-to number retrieved during call setup to the destination • redirectReason—the type of redirection – rr_cfb—CF-busy – rr_cfnr—CF-no answer – rr_cd—CD-call deflection – rr_cfu—CF-unconditional • redirectCount—number of call diversions that have occurred • originalDest—original called number
Mode	Read
Scope	ev_setup_done
Return Type	String
Direct Mapping	None

evt_service_control

Description	Returns the service control indexed by *<index>*, with *<index> 1* being the first service control field.
Syntax	**infotag get evt_service_control <index>**
Mode	Read
Scope	ev_address_resolved
Return Type	String The string content is application dependent. The format of the content are agreed upon between the application and the route entity. **Note** The application processes the service descriptor fields. Neither the gatekeeper nor the gateway interprets the contents of the service descriptors.
Direct Mapping	None

evt_service_control_count

Description	Returns the number of service control fields.
Syntax	**infotag get evt_service_control_count**
Mode	Read
Scope	ev_address_resolved
Return Type	Number
Direct Mapping	None

evt_status

Description	Returns the status of the event received. This info-tag is valid only in the scope of the function handling the event. For a list of possible statuses, see the "Status Codes" section on page 5-6.
Syntax	**infotag get evt_status**
Mode	Read
Scope	ev_setup_done ev_collectdigits_done ev_media_done ev_disconnected ev_authorize_done ev_authenticate_done ev_vxmldialog_done ev_subscribe_done ev_unsubscribe_done ev_unsubscribe_indication
Return Type	String
Direct Mapping	None

Examples	`proc act_SubscribeDone { } {`
	`  puts "***** act_Subscribe : SUBSCRIPTION DONE received"`
	`  set sub_id [infotag get evt_subscription_id]`
	`  set status [infotag get evt_status]`
	`  puts "*** act_subscribe: subscription status=$status"`
	`  if {$status == "su_000"} {`
	`  puts "\n Subscription accepted."`
	`  return`
	`  } else if {$status == "su_001"} {`
	`  puts "\n Subscription is pending"`
	`  return`
	`  } else {$status == "su_002"} {`
	`  # subscription request has failed; close the subscription`
	`  puts "\nSubscription request has failed."`
	`  set status_text [infotag get evt_status_text]`
	`  puts "text is: $status_text\n"`
	`  subscription close $sub_id`
	`  }`
	`}`

Usage Notes	Return codes indicate some type of failure has occurred; therefore, the script should be written to handle such errors accordingly.

evt_status_text

Description	Retrieves failure information associated with an event.
Syntax	**infotag get evt_status_text**
Mode	Read
Scope	ev_subscribe_done, ev_unsubscribe_done, ev_unsubscribe_indication
Return Type	String with failure information, if any.
Direct Mapping	None

Example	`set sub_id [infotag get evt_subscription_id]`
	`set status [infotag get evt_status]`
	`puts "*** act_subscribe: subscription status=$status"`
	`  if {$status == "su_002"} {`
	`  # subscription request has failed; close the subscription`
	`  puts "\nSubscription request has failed."`
	`  set status_text [infotag get evt_status_text]`
	`  puts "text is: $status_text\n"`
	`  subscription close $sub_id`
	`  }`

evt_subscription_id

Description	Retrieves the subscription id associated with events related to a subscription or a notification.
Syntax	**infotag get evt_subscription_id**
Mode	Read
Scope	ev_subscribe_done, ev_notify, ev_subscribe_cleanup, ev_unsubscribe_indication
Return Type	Subscription ID
Direct Mapping	None
Examples	`set sub_id [infotag get evt_subscription id]`
Usage Notes	None

evt_timer_name

Description	Retrieves the name associated with the expired named_timer.
Syntax	**infotag get evt_timer_name**
Mode	Read
Scope	ev_named_timer
Return Type	String
Direct Mapping	None

evt_transfer_info

Description	Returns transfer information from a transfer request event.
Syntax	**infotag get evt_transfer_info** {*transferBy* | *transferDest* | *consultID*}
Mode	Read
Scope	ev_transfer_request
Return Type	String
Direct Mapping	None

evt_vxmlevent

Description	Returns a string containing the VXML event that was thrown. These events are generally of the form *vxml.**. Events thrown from the dialog markup, or the document using the VXML *sendevent object* extension, are of the form vxml.dialog.*. For more information on sendevent objects, refer to SendEvent Object, page 1-8. Events thrown by the system due to some event, such as the vxml document executing a <disconnect/> tag, are of the form vxml.session.*.
Syntax	**infotag get evt_vxmlevent**
Mode	Read
Scope	ev_vxmldialog_done ev_vxmldialog_event
Return Type	String
Direct Mapping	None

evt_vxmlevent_params

Description	Retrieves parameters that may come with an event. This info-tag clears the array variable and fills it with the parameter values indexed by the parameter names in the param option of the sendevent object tag. Parameters can also be returned through the <exit/> tag with a namelist attribute. For more information on sendevent objects, refer to SendEvent Object, page 1-8. In either case, if the namelist contains an audio clip variable, it is made available to the Tcl script as a parameter with a string value containing the *ram:// uri* form for the audio clip. The info tag returns a space-separated list of indexes that were added to the return array variable passed as a parameter to the information tag.
Syntax	**infotag get evt_vxmlevent_params** <*array-variable-name*>
Mode	Read
Scope	ev_vxmldialog_done ev_vxmldialog_event
Return Type	String Parameter: array-variable-name
Direct Mapping	None

gtd_attr_exists

Description	Used to determine if an attribute instance exists in a GTD message.
Syntax	**infotag get gtd_attr_exists** <*gtd_handle*><*attr_instance*> • <*gtd_handle*>—Name of the GTD handle from which the application wants to check the existence of a GTD attribute instance. • <*attr_instance*>—This parameter is of the form <*attr_name*>, <*attr_instance*>. <*attr_instance*> can be specified with a value of -1, which means "don't care."
Mode	Read
Scope	Global
Return Type	Boolean (1=true; 0=false)
Direct Mapping	None

last_command_handle

Description	Retrieves the last command handle.
Syntax	**infotag get last_command_handle**
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

leg_all

Description	Returns or maps to one or more call legs. This is the union of leg_incoming and leg_outgoing.
Syntax	**infotag get leg_all**
Mode	Read
Scope	Global
Return Type	Number list
Direct Mapping	Legs

leg_ani

Description	Returns the ANI field of CallInfo.
Syntax	**infotag get leg_ani** [*legID*] If no leg ID is specified, this info-tag returns the ANI field of the first incoming call leg. Not specifying a leg ID works only if there is at least one incoming call leg. If a leg ID is specified, this info-tag returns the ANI field of that call leg. If the call leg is not valid, the script terminates with error output.
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

leg_ani_pi

Description	Gets the calling number presentation indication value.
Syntax	**infotag get leg_ani_pi**
Mode	Read
Scope	Global
Return Type	Number list Values retrieved could be one of the following: 1—presentation_allowed 2—presentation_restricted 3—number_lost_due_to_interworking 4—reserved_value 5—not_present (denotes that the Calling Number IE is absent in the incoming signaling message.
Direct Mapping	None

leg_ani_si

Description	Gets the calling number screening indication value.
Syntax	**infotag get leg_ani_si**
Mode	Read
Scope	Global
Return Type	Number list Values retrieved could be one of the following: 1—usr_provided_unscreened 2—usr_provided_screening_passed 3—usr_provided_screening_failed 4—network_provided 5—not_present (denotes that the Calling Number IE is absent in the incoming signaling message.
Direct Mapping	None

leg_dn_tag

Description	Returns the DN field of call info. In an Ephone-initiated call, it carries the DN tag of the calling party.
Syntax	**infotag get leg_dn_tag** *legID*
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

leg_dnis

Description	Returns the DNIS field of CallInfo.
Syntax	**infotag get leg_dnis** [*legID*] If no leg ID is specified, this info-tag returns the DNIS field of the first incoming call leg. Not specifying a leg ID works only if there is at least one incoming call leg. If a leg ID is specified, this info-tag returns the DNIS field of that call leg. If the call leg is not valid, the script terminates with error output.
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

leg_display_info

Description	Returns the display_info field of call info. In an Ephone-initiated call, this field contains the name of the calling party.
Syntax	**infotag get leg_display_info** *legID*
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

leg_guid

Description	Returns the GUID o a leg.
Syntax	**infotag get leg_guid** [*legID*] If legID is not specified, returns the GUID of the first incoming leg.
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

leg_incoming

Description	Returns or maps to one or more incoming call legs.
Syntax	**infotag get leg_incoming**
Mode	Read
Scope	Global
Return Type	Number list
Direct Mapping	Legs

leg_incoming_guid

Description	Returns the incoming GUID of a leg.
Syntax	**infotag get leg_incoming_guid** [legID] If legID is not specified, returns the GUID of the first incoming leg.
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

leg_inconnection

Description	Gets a list of legs that are part of this connection. The info-tag parameter maps to just one connection.
Syntax	**infotag get inconnection** {*connID* | *info-tag*}
Mode	Read
Scope	Global
Return Type	Number list
Direct Mapping	Legs

leg_isdid

Description	Returns the DID field of CallInfo. This is a Boolean field (1 and 0) that reflects the FinalDestination flag of the call leg.
Syntax	**infotag get leg_isdid** [*legID*] If no leg ID is specified, this info-tag returns the DID field of the first incoming call leg. Not specifying a leg ID works only if there is at least one incoming call leg. If a leg ID is specified, this info-tag returns the DID field of that call leg. If the call leg is not valid, the script terminates with error output.
Mode	Read
Scope	Global
Return Type	Boolean (1 = true; 0 = false)
Direct Mapping	None

leg_outgoing

Description	Returns or maps to one or more outgoing call legs.
Syntax	**infotag get leg_outgoing**
Mode	Read
Scope	Global
Return Type	Number list
Direct Mapping	Legs

leg_password

Description	If no leg ID is specified, this info-tag returns the password field of the first incoming call leg. Not specifying a leg ID works only if there is at least one incoming call leg. If a leg ID is specified, this info-tag returns the password field of that call leg. If the call leg is not valid, the script terminates with error output.
Syntax	**infotag get leg_password** [*legID*]
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

leg_proto_headers

Description	Provides access to headers associated with the protocol being used.
Syntax	**infotag get leg_proto_headers** [<*attribute-name*>] [*legID*] <*attribute-name*>—Name of the header to get.
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

Example	`set AccountInfo [infotag get leg_proto_headers AccountInfo]`
	`set inviteSubject [infotag get evt_proto_headers "Subject"]`
	`set inviteFrom [infotag get leg_proto_headers "From" leg_incoming]`
	The following command returns all headers received from the incoming Invite message in a concatenated string. Each header av-pair is separated by a '&':
	`set allHeaders [infotag get evt_proto_headers]`

Usage Notes	• This information tag allows the accessing of SIP headers from VXML documents or TCL IVR 2.0 scripts. • Currently, only access to headers in SIP invite, subscribe, notify and H.323 setup messages are supported. • If <attribute-name> is not specified, all headers are returned in a concatenated string, with each header separated by a "&" symbol. • If legID is not provided, the first incoming leg is applied.

leg_rdn

Description	Gets the redirect number from the first incoming leg.
Syntax	**infotag get leg_rdn**
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

leg_rdn_pi

Description	Gets the redirect number presentation indication value.
Syntax	**infotag get leg_rdn_pi**
Mode	Read
Scope	Global
Return Type	Number list Values retrieved could be one of the following: 1—presentation_allowed 2—presentation_restricted 3—number_lost_due_to_interworking 4—reserved_value 5—not_present (denotes that the Redirect Number IE is absent in the incoming signaling message.
Direct Mapping	None

leg_rdn_si

Description	Gets the redirect number screening indication value.
Syntax	**infotag get leg_rdn_si**
Mode	Read
Scope	Global
Return Type	Number list Values retrieved could be one of the following: 1—usr_provided_unscreened 2—usr_provided_screening_passed 3—usr_provided_screening_failed 4—network_provided 5—not_present (denotes that the Redirect Number IE is absent in the incoming signaling message.
Direct Mapping	None

leg_redirect_cnt

Description	Retrieves redirection count information from the first incoming call leg or for a leg if callid is specified.
Syntax	**infotag get leg_redirect_cnt**
Mode	Read
Scope	Global
Return Type	Number. Values retrieved between 0–7.
Direct Mapping	None

leg_remoteipaddress

Description	Returns the remote IP address of the endpoint from which the call is received. If the IP address is not available, an empty string is returned.
Syntax	**infotag get leg_remoteipaddress** <leg-id>
Mode	Read
Scope	Global
Return Type	String (ip address)
Direct Mapping	None

leg_remote_media_ip_address

Description	Returns the remote media IP address of the endpoint. If the IP address is not available, an empty string is returned.
Syntax	**infotag get leg_remote_media_ip_address** <leg_id>
Mode	Read
Scope	Global
Return Type	String (ip address)
Direct Mapping	None

leg_remote_signaling_ip_address

Description	Returns the remote signaling IP address of the endpoint. If the IP address is not available, an empty string is returned.
Syntax	**infotag get leg_remote_signaling_ip_address** <leg_id>
Mode	Read
Scope	Global
Return Type	String (ip address)
Direct Mapping	None

leg_rgn_noa

Description	Gets the redirect number nature of address value.
Syntax	**infotag get leg_rgn_noa**
Mode	Read
Scope	Global
Return Type	Number Values retrieved could be one of the following: 00—Unknown, number present 01—Unknown, number absent, presentation restricted 02—Unique subscriber number 03—Nonunique subscriber number 04—Unique national (significant) number 05—Nonunique national number 06—Unique international number 07—Nonunique international number 08—Network specific number 09—Nonsubscriber number 10—Subscriber number, operator requested 11—National number, operator requested 12—International number, operator requested 13—No number present, operator requested 14—No number present, cut through call to carrier 15—950+ call from local exchange carrier public station, hotel/motel or non-exchange access end office 16—Test line test code 17—Unique 3 digit national number 18—Credit card 19—International inbound 20—National or international with carrier access code included 21—Cellular - global ID GSM 22—Cellular - global ID NWT 900 23—Cellular - global ID autonet 24—Mobile (other) 25—Ported number 26—VNET 27—International operator to operator outside WZ1 28—International operator to operator inside WZ1 29—Operator requested - treated 30—Network routing number in national (significant) format 31—Network routing number in network specific format 32—Network routing number concatenated with called directory number 33—Screened for number portability 34—Abbreviated number
Direct Mapping	None

Note This infotag has been provided as an interim mechanism for accessing specific signaling information. It may be obsoleted in a future IOS release when an alternate method of accessing this information is made

available.

leg_rgn_npi

Description	Returns the redirect number numbering plan indicator value.
Syntax	**infotag get leg_rgn_npi**
Mode	Read
Scope	Global
Return Type	Number Values retrieved could be one of the following: 1—ISDN numbering plan 2—Data numbering plan 3—Telex numbering plan 4—Private numbering plan 5—National 6—Maritime mobile 7—Land mobile 8—ISDN mobile 252—Unknown
Direct Mapping	None

Note This infotag has been provided as an interim mechanism for accessing specific signaling information. It may be obsoleted in a future IOS release when an alternate method of accessing this information is made available.

leg_rgn_num

Description	Returns the redirect number address.
Syntax	**infotag get leg_rgn_num**
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

Note This infotag has been provided as an interim mechanism for accessing specific signaling information. It may be obsoleted in a future IOS release when an alternate method of accessing this information is made available.

leg_rgn_pi

Description	Returns the redirect number presentation indicator value.
Syntax	**infotag get leg_rgn_pi**
Mode	Read
Scope	Global
Return Type	Number Values retrieved could be one of the following: 0—Unknown 1—Presentation allowed 2—Presentation not allowed 3—Address not available
Direct Mapping	None

Note This infotag has been provided as an interim mechanism for accessing specific signaling information. It may be obsoleted in a future IOS release when an alternate method of accessing this information is made available.

leg_rgn_si

Description	Returns the redirect number screening indicator value.
Syntax	**infotag get leg_rgn_si**
Mode	Read
Scope	Global
Return Type	Number Values retrieved could be one of the following: 1—User provided not screened 2—User provided screening passed 3—User provided screening failed 4—Network provided 252—Unknown
Direct Mapping	None

Note This infotag has been provided as an interim mechanism for accessing specific signaling information. It may be obsoleted in a future IOS release when an alternate method of accessing this information is made available.

leg_settlement_time

Description	Returns the minimum of the OSP settlement time (in seconds) associated with the list of specified legs.
Syntax	**infotag get leg_settlement_time** {**legID** | **info-tag**} [**minimum**] If you specify minimum, this returns the minimum of the OSP settlement time of the list of legs and the value of the AAA AV-pair creditTime. This AAA AV-pair creditTime was returned by a previous **aaa authorize** command. If all credit times are uninitialized, "uninitialized" is returned. If all have unlimited time, or if one is uninitialized and the others have unlimited time, "unlimited" is returned.
Mode	Read
Scope	Global
Return Type	Number
Direct Mapping	None

leg_source_carrier_id

Description	Retrieve the source carrier ID.
Syntax	**infotag get leg_source_carrier_id**
Mode	Read
Scope	Global
Return Type	None
Direct Mapping	None

leg_subscriber_type

Description	Returns the subscriber type.
Syntax	**infotag get leg_subscriber_type**
Mode	Read
Scope	Global
Return Type	None
Direct Mapping	None

leg_suppress_outgoing_auto_acct

Description	When set, the service provider module does not automatically generate an accounting packet on the outgoing call leg.
Syntax	**infotag get leg_suppress_outgoing_auto_acct** **infotag set leg_suppress_outgoing_auto_acct**
Mode	Read/write
Scope	Global
Return Type	None for set Boolean (0 | 1) for get
Direct Mapping	Leg

leg_target_carrier_id

Description	Set the target carrier ID.
Syntax	**infotag set leg_target_carrier_id**
Mode	Write
Scope	Global
Return Type	String
Direct Mapping	None

leg_tdm_hairpin

Description	Enables TDM hairpinning on the incoming call leg. Copies the TDM hairpin value to the outgoing leg and enables it for TDM hairpinning.
Syntax	**infotag set leg_tdm_hairpin** [legID | info-tag] [enable | disable]
Mode	Write
Scope	Global
Return Type	None
Direct Mapping	None

leg_type

Description	Returns the interface type of the specified leg. If the *detail* parameter is specified, this returns a detailed string that includes the interface subtype of the specified leg. The *detail* parameter is optional. The information tag used to specify the leg should map to just one leg.
Syntax	**infotag get leg_type** [detail] {legID | info-tag}
Mode	Read
Scope	Global

Return Type	String Possible return values are: telephony voip mmoip voatm vofr unknown none Possible return values with the *detail* parameter are: voip-h323 voip-sip tele-analog-em tele-analog-fxo tele-analog-fxs tele-analog-efxs tele-analog-efxo tele-digital-isdn tele-digital-cas tele-digital-bri tele-digital-r2 msp-doc msp-fax unknown
Direct Mapping	None

leg_username

Description	If no leg ID is specified, this info-tag returns the username field of the first incoming call leg. Not specifying a leg ID works only if there is at least one incoming call leg. If a leg ID is specified, this info-tag returns the username field of that call leg. If the call leg is not valid, the script terminates with error output.
Syntax	**infotag get leg_username** [*legID*]
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

med_backup_server

Description	Returns or sets the backup server. This is applicable for RTSP-based prompts. If the script attempts to play a prompt using a URL and the URL fails, it tries to replay the URL from a list of backup servers by replacing the server portion of the URL. For example, if the script tries (but fails) to play a prompt from: rtsp://www.cisco.com:5554/audiofiles/english/anounce.au and the backup server 0 is configured as: rtsp://www.real.com/cisco/ then the backup URL attempted is: rtsp://www.real.com/cisco/audiofiles/english/anounce.au A maximum of two (0 and 1) backup servers can be configured. This info-tag applies only to streams on which you have not played any prompts and is typically used in the one-time initialization section of the script.
Syntax	**infotag get med_backup_server** *index* **infotag set med_backup_server** *index server-URL*
Mode	Read/Write
Scope	Global
Return Type	String
Direct Mapping	None

med_language

Description	Returns or sets the current active language for media playout. This info-tag returns the language index or the language prefix (depending on whether prefix is specified) for the currently active language.
Syntax	**infotag get med_language** [**prefix**] **infotag set med_language** [*index* | **prefix** *prefix*]
Mode	Read/Write
Scope	Global
Return Type	String/Number
Direct Mapping	None

med_language_map

Description	Returns or sets the mapping between the language index and the language prefix. This info-tag returns the language index or the language prefix (depending on whether prefix is specified) for the currently active language.
Syntax	**infotag get med_language_map** [*index* | **prefix** *prefix*] **infotag set med_language_map** *index prefix*
Mode	Read/Write
Scope	Global
Return Type	String/Number
Direct Mapping	None

med_location

Description	Returns or sets the language locations for all the languages the script uses. The language prefix, category, and location are the same as those configurable from the Cisco IOS command line interface (CLI).
Syntax	**infotag get med_location** *prefix category.* Valid category values are 1, 2, 3, 4. **infotag set med_location** *prefix category location.* Category 0 can be used to set all 1–4 categories.
Mode	Read/Write
Scope	Global
Return Type	String
Direct Mapping	None

med_total_languages

Description	Returns the total number of languages configured.
Syntax	**infotag get med_total_languages**
Mode	Read
Scope	Global
Return Type	Number
Direct Mapping	None

media_timer_factor

Description	Sets the receive-rtcp timer. The new value is used within the scope of the script and does not change the gateway configuration.
Syntax	**infotag set media_timer_factor** <*timer_factor*> *timer_factor*—An integer between 2–1000. The value is a multiple of the RTCP transmission interval. A value of 5 is recommended. **Note** If a value outside the range of 2–1000 is used, the script receives an error message.
Mode	Write
Scope	None
Return Type	None
Direct Mapping	None
Example	infotag set media_timer_factor 5

mod_all_handles

Description	Retrieves a list of all instances running on the gateway. The returned handle can be used as an argument for **handoff** and **sendmsg** commands.
Syntax	**infotag get mod_all_handles**
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None
Examples	`set all_handles [infotag get mod_all_handles]`
Usage Notes	• Handles are ASCII strings that are only valid within a gateway. A handle for a session on another gateway cannot be used to send a message to that gateway. • The format of a handle is not designed to be parsed or printed by a TCL script. The handle is used internally by a sendmsg or handoff command.

mod_handle

Description	Retrieves the handle of the currently running application session. The returned handle can be used as an argument for **handoff** and **sendmsg** commands.
Syntax	**infotag get mod_handle**
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

Examples	`set id [infotag get mod_handle]`
	
	`set my_name new_customer`
	`set recorded_name ([infotag get mod_handle]) $my_name`

Usage Notes	• Handles are ASCII strings that are only valid within a gateway. A handle for a session on another gateway cannot be used to send a message to that gateway. • The format of a handle is not designed to be parsed or printed by a TCL script. The handle is used internally by a **sendmsg** or **handoff** command. • Returns “unavailable” if the service is not running.

mod_handle_service

Description	Retrieves the handle of the named service.
Syntax	**infotag get mod_handle_service** <*service*>
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None
Example	`set serv_hndl [infotag get mod_handle_service]`
Usage Notes	• Returns “unavailable” if the named service is not running. • The returned handle can be used as an argument for **handoff** and **sendmsg** commands.

set iec

Description	Sets the Internal Error Code (IEC) before issuing a **call close** or **leg disconnect** command. **Note** Using the *set iec* information tag after specifying the IEC with the **leg disconnect** -<*iec*> command causes duplicate IECs to be associated with the call leg.
Syntax	**infotag set iec** <*legID* | *info-tag*> <*iec*> <*iec*>—Can be one of the following arguments: • *media_done_err*—Indicates that the script is terminating the call because of an error status returned by the ev_media_done event. • *collectdigits_done_err*—Indicates that the script is terminating the call because of an error status returned by the ev_collectdigits_done event. • *authenticate_done_err*—Indicates that the script is terminating the call because of an error status returned by the ev_authenticate_done_event. • *authorize_done_err*—Indicates that the script is terminating the call because of an error status returned by the ev_authorize_done event. • *media_inactivity_err*—Disconnects the call when media inactivity is detected and reported. • *accounting_conn-err*—Indicates the script detects that connectivity to the accounting server is lost.
Mode	Write
Scope	Global
Return Type	String
Direct Mapping	None

subscription_context

Description	Retrieves the subscription context information associated with the subscription ID.
Syntax	**infotag get subscription_context** <*subscription_id*> *[attribute]* • *subscription_id*—ID associated with the subscription • *attribute*—The attribute within the context information.
Mode	Read
Scope	Global
Return Type	String containing the context.
Direct Mapping	None
Example	`set accountNumber [infotag get subscription_context $MySubID accountNumber]`
Usage Notes	If the *attribute* argument is missing, a string containing all attributes and values is returned. Each av-pair is escaped by the "#" character.

subscription_info

Description	Retrieves the subscription information associated with a subscription ID. Retrievable subscription attributes are url, event, expirationTime, subscription_context, and notificationReceiver.
Syntax	**infotag get subscription_info** <*subscription_id*> <*attribute*> • *subscription_id*—ID associated with the subscription • *attribute*—The attribute to access.
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None
Example	`set exp_time [infotag get subscription_info $sub_id expirationTime]`
Usage Notes	None

subscription_server_ipaddress

Description	Retrieves the IP address of the subscription server.
Syntax	**infotag get subscription_server_ipaddress** <*subscription_id*> • *subscription_id*—ID associated with the subscription
Mode	Read
Scope	Global
Return Type	String containing the IP address of the server.
Direct Mapping	None
Example	`set ipaddr [infotag get subscription_server_ipaddress $sub_id]`
Usage Notes	None

sys_version

Description	Returns the version of the Tcl IVR API.
Syntax	**infotag get sys_version**
Mode	Read
Scope	Global
Return Type	String
Direct Mapping	None

Events and Status Codes

This chapter describes events received and status codes returned by Tcl IVR scripts. This chapter includes the following topics:

Events

The following events can be received by the Tcl IVR script. Any events received that are not included below are ignored.

Event	Description
ev_accounting_status_ind	Received when the method list or server group is marked unreachable. The accounting status and method list can be derived using infotag get evt_aaa_status_info [attribute-name].
ev_address_resolved	List of endpoint addresses.
ev_alert	An intermediate event generated by the **leg setup** or **leg setup_continue** commands to set up a call. If specified in the callinfo parameter, *notifyEvents,* the script receives an **ev_alert** message once the destination endpoint is successfully alerted. The script running in the transferee gateway could then disconnect the leg towards the transferring endpoint. If this event is an intercepted event, the application needs to use the **leg setup_continue** command to allow the system to continue with the setup.
ev_any_event	A special wildcard event that can be used in the state machine to represent any event that might be received by the script.
ev_authorize_done	Confirms the completion of the **aaa authorize** command. You can use the evt_status info-tag to determine the authorization status (whether it succeeded or failed).
ev_authenticate_done	Confirms the completion of the authentication command. You can use the evt_status info-tag to determine the authentication status (whether it succeeded or failed).
ev_call_timer0	Indicates that the call-level timer expired.

Event	Description
ev_collectdigits_done	Confirms the completion of the **leg collectdigits** command on the call leg. You can then use the evt_status info-tag to determine the status of the command completion. You can use the evt_dcdigits info-tag to retrieve the collected digits.
ev_connected	An intermediate event generated by the **leg setup** or **leg setup_continue** commands to set up a call. If the callinfo paramater, *notifyEvents*, is specified, the script receives an *ev_connected* message when the system receives a connect event from the destination switch. If this event is an intercepted event, the application needs to use the **leg setup_continue** command to allow the system to continue with the setup.
ev_consult_request	Indicates a call-transfer consultation-id request from an endpoint.
ev_consult_response	Indicates a response to the **leg consult request** command. For return codes, see Consult Status under Status Codes.
ev_consultation_done	Indicated the completion of a **leg consult response** command. For return codes, see Consult Response under Status Codes.
ev_create_done	Confirms the completion of the **connection create** command. You can use the evt_connection info-tag to determine the ID of the completed connection.
ev_destroy_done	Confirms the completion of the **connection destroy** command. You can use the evt_connection info-tag to determine the ID of the connection that was destroyed.
ev_digit_end	Indicates that a digit key is pressed and released. You can use the evt_digit info-tag to determine which digit was pressed. You can use the evt_digit_duration info-tag to determine how long (in seconds) the digit was pressed. This can be used to detect long pounds or long digits.
ev_disconnect_done	Indicates that the call leg has been cleared.
ev_disconnected	Indicates that one of the call legs needs to disconnect. On receiving this event, the script must issue a **leg disconnect** on that call leg. You can use the evt_legs info-tag to determine which call leg disconnected.
ev_disc_prog_ind	Indicates a DISC/PI message is received at a call leg.
ev_facility	Indicates a response to a **leg facility** command.
ev_feature	Indicates a feature event received by the script. The script can use the **set evt_feature_report** information tag to enable or disable the feature events to be intercepted. When the script receives an *ev_feature* event, it can use the **get evt_feature_type** information tag to retrieve the feature type string.
ev_grab	Indicates that an application that called this script is requesting that the script return the call leg. The script receiving this event can clean up and return the leg with a **handoff return** command. Whether this is done is at the discretion of the script receiving the ev_grab event.

Event	Description
ev_hookflash	Indicates a hook flash (such as a quick onhook-offhook in the middle of a call), assuming that the underlying platform or interface supports hook flash detection. It is received by the TCL script when the user presses a hookflash.
ev_handoff	Indicates that the script received one or more call legs from another application. When the script receives this event, you can use the evt_legs and the evt_connections info-tags to obtain a list of the call legs and connection IDs that accompanied the ev_handoff event.
ev_leg_timer	Indicates that the leg timer expired. You can use the evt_legs info-tag to determine which leg timer expired.
ev_media_activity	Indicates the detection of an active call. It is generated when the RTP and RTCP packets are transmitted again after a period of media inactivity.
ev_media_done	Indicates that the prompt playout either completed or failed. You can use the evt_status info-tag to determine the completion status.
ev_media_inactivity	Indicates the detection of an inactive call. It is generated if the RTP and RTCP packets are not received during a specified time period. The time period is specified by the CLI **ip rtcp report interval** and **timer receive-rtcp**.
ev_named_timer	Received when a named_timer expires. The name of the named_timer can be derived by using the *get evt_timer_name* information tag.
ev_proceeding	An intermediate event generated by the **leg setup** or **leg setup_continue** commands to set up a call. If the callinfo paramater, *notifyEvents*, is specified, the script receives an *ev_proceeding* message when the system receives a proceeding event from the remote end. If this event is an intercepted event, the application needs to use the **leg setup_continue** command to allow the system to continue with the setup.
ev_progress	An intermediate event generated by the **leg setup** or **leg setup_continue** commands to set up a call. If the callinfo paramater, *notifyEvents*, is specified, the script receives an *ev_progress* message when the system receives a progress event from the destination switch. If this event is an intercepted event, the application needs to use the **leg setup_continue** command to allow the system to continue with the setup.
ev_returned	Indicates that a call leg that was sent to another application (using **handoff callappl**) has been returned. This event can be accompanied by one or more call legs that were created by the called application. When the script receives this event, you can use the evt_legs and the evt_connections info-tags to obtain a list of the call legs and connection IDs that accompanied the ev_returned event. You can use the evt_iscommand_done info-tag to verify that all of the call legs sent have been accounted for, meaning that the **handoff callappl** command is complete.

Event	Description
ev_setup_done	Indicates that the **leg setup** command has finished. You can then use the evt_status info-tag to determine the status of the command completion (whether the call was successfully set up or failed for some reason).
ev_setup_indication	Indicates that the system received a call. This event and the ev_handoff event are the events that initiate an execution instance of a script.
ev_synthesizer	Indicates the completion of a **media play** command.

ev_tone_detected	Signifies the detection of the requested tone. This event is generated, at most, once after a **leg tonedetect enable** command is issued. Tone detection is disabled after this event arrives. Use the *evt_status* information tag to determine the detected tone. See Status Codes, page 5-6, for possible tone detect status values.
	Example:
	`set fsm(WAIT_FOR_CNG, ev_tone_detected) "act_process_td_event, same_state")`
	`proc act_process_td_event { } { set Tone1 [infotag get evt_status] if { $Tone1 == "td_003" } # Do stuff here }`

ev_transfer_request	Indicates a call transfer from an endpoint to the application.
ev_transfer_status	An intermediate event generated by the **leg setup** command. If specified in the callinfo parameter, *notifyEvents*, the script receives an **ev_trasfer_status** message. The *ev_status* information tag would then contain the status value of the call transfer.
ev_vxmldialog_done	Received when the VXML dialog completes. This could be because of a VXML dialog executing an <exit/> tag or interpretation completing the current document without a transition to another document. The dialog could also complete due to an interpretation failure or a document error. This completion status is also available through the evt_status info-tag.
ev_vxmldialog_event	Received by the Tcl IVR application when the VXML dialog initiated on a leg executes a sendevent object tag. The VXML subevent name is available through the evt_vxmlevent info-tag. All events thrown from the dialog markup are of the form vxml.dialog.*. All events generated by the system—perhaps as an indirect reaction to the VXML document executing a certain tag or throwing a certain event—like the dialog completion event are of the form vxml.session.*.
ev_msg_indication	Signals an incoming message.
ev_session_indication	Signals the start of a new session.
ev_session_terminate	Stop the current TCL session.
ev_subscribe_done	Subscription request completed.
ev_unsubscribe_done	Unsubscribe request completed.
ev_unsubscribe_indication	Server terminated the subscription.
ev_notify	Notify indication received.
ev_notify_done	Notification request completed

Event	Description
ev_subscribe_cleanup	Received when a 'clear subscription <session id | all' CLI is executed.
ev_returned	Received when another application instance returns the call leg.
ev_grab	If a user stops a TCL session that has already handed a call off to a second session, the session sends an ev_grab event to the second session. If the second session returns the call leg, the first session cleans up. If the second session does not return the call leg, the first session stops executing, but does not clean up completely until the second session disconnects the call leg, or returns it.

Usage Notes:

- Scripts must check the return status for the *ev_subscribe_done* event. This event indicates that a response is received from server. A return code of **su**_*000* indicates that a positive response has been received. A return code of **su**_*002* or above indicates a negative response from the server.
- A subscription is complete only when an *ev_subscribe_done* event and the first notification from server are received. An application should close its instances only after making sure the subscription is complete.
- The script receives an *ev_unsubscribe_indication* event when the server terminates the subscription. The script can access header and content information associated with this event.
- If the subscription timer expires, the script receives an *ev_unsubscribe_indication* event with a status code of **ui**_*003*. Since this is an internal event, there is no header or content information associated with this event.
- When an *ev_subscribe_cleanup* event is received, the script must close the subscription. If no response is received within 5 seconds, the infrastructure closes the subscription. Make sure the script handles this event.
- If the instance making the subscription is already closed and an *ev_notify_indication*, *ev_subscribe_cleanup*, or *ev_unsubscribe_indication* event is received, a new instance is created and the event is handed to it.

Status Codes

The evt_status info-tag returns a status code for the event received. This sections lists the possible status codes and their meaning.

Status codes are grouped according to function. The first two characters of the status code indicate the grouping.

- au—Authentication status
- ao—Authorization status
- cd—Digit collection status
- cr—Consult response
- cs—Consult status
- di— Disconnect cause
- fa—Facility
- ft—Feature type
- ls—Leg setup status
- ms—Media status
- td—Tone detect
- ts—Transfer status
- vd—Voice dialog completion status

Authentication Status

Authentication status is reported in **au**_*xxx* format:

Value for *xxx*	Description
000	Authorization was successful.
001	Authorization error.
002	Authorization failed.

Authorization Status

Authorization status is reported in **ao**_*xxx* format:

Value for *xxx*	Description
000	Authorization was successful.
001	Authorization error.
002	Authorization failed.

Digit Collection Status

Digit collection status is reported in **cd**_*xxx* format:

Value for *xxx*	Description
001	The digit collection timed out, because no digits were pressed and not enough digits were collected for a match.
002	The digit collection was aborted, because the user pressed an abort key.
003	The digit collection failed, because the buffer overflowed and not enough digits were collected for a match.
004	The digit collection succeeded with a match to the dial plan.
005	The digit collection succeeded with a match to one of the patterns.
006	The digit collection failed because the number collected was invalid.
007	The digit collection was terminated because an ev_disconnected event was received on the call leg.
008	The digit collection was terminated because an ev_grab event was received on the call leg.
009	The digit collection successfully turned on digit reporting to the script.
010	The digit collection was terminated because of an unsupported or unknown feature or event.

Consult Response

Feature type is reported in **cr**_*xxx* format:

Value for *xxx*	Description
000	Success
001	Failed, invalid state
002	Failed, timeout
003	Failed, abandon
004	Failed, protocol error

Consult Status

Feature type is reported in **cs**_*xxx* format:

Value for *xxx*	Description
000	Consultation success, consult-id available
001	Consultation failed, request timeout
002	Consultation failed
003	Consultation failed, request rejected
004	Consultation failed, leg disconnected
005	Consultation failed, operation unsupported

Disconnect Cause

Disconnect causes use the format **di**_*xxx* where *xxx* is the Q931 cause code. Possible values are:

Value for *xxx*	Description
000	Uninitialized
001	Unassigned number
002	No route to the transit network
003	No route to the destination
004	Send information tone
005	Misdialed trunk prefix
006	Unacceptable channel
007	Call awarded
008	Preemption
009	Preemption reserved
016	Normal
017	Busy
018	No response from the user
019	No answer from the user
020	Subscriber is absent
021	Call rejected
022	Number has changed
026	Selected user is clearing
027	Destination is out of order
028	Invalid number
029	Facility rejected
030	Response to status inquiry
034	No circuit available
035	Requested VPCI VCI is not available
036	VPCI VCI assignment failure
037	Cell rate is not available
038	Network is out of order
039	Permanent frame mode is out of service
040	Permanent frame mode is operational
041	Temporary failure
042	Switch is congested
043	Access information has been discarded
044	No required circuit
045	No VPCI VCI is available
046	Precedence call blocked

Value for *xxx*	Description
047	No resource available
048	DSP error
049	QoS is not available
050	Facility is not subscribed
053	Outgoing calls barred
055	Incoming calls barred
057	Bearer capability is not authorized
058	Bearer capability is not available
062	Inconsistency in the information and class
063	Service or option not available
065	Bearer capability is not implemented
066	Change type is not implemented
069	Facility is not implemented
070	Restricted digital information only
079	Service is not implemented
081	Invalid call reference value
082	Channel does not exist
083	Call exists and call ID in use
084	Call ID in use
085	No call suspended
086	Call cleared
087	User is not in CUG
088	Incompatible destination
090	CUG does not exist
091	Invalid transit network
093	AAL parameters not supported
095	Invalid message
096	Mandatory information element (IE) is missing
097	Message type is not implemented
098	Message type is not compatible
099	IE is not implemented
100	Invalid IE contents
101	Message in incomplete call state
102	Recovery on timer expiration
103	Nonimplemented parameter was passed on
110	Unrecognized parameter message discarded
111	Protocol error

Value for *xxx*	Description
127	Internetworking error
128	Next node is unreachable
129	Holst Telephony Service Provider Module (HTSPM) is out of service
160	DTL transit is not my node ID

Facility

Leg setup requesting address resolution status is reported in **fa**_*xxx* format:

Value for *xxx*	Description
000	supplementary service request succeeded
003	supplementary service request unavailable
007	supplementary service was invoked in an invalid call state
009	supplementary service was invokes in a non-incoming call leg
010	supplementary service interaction is not allowed
050	MCID service is not subscribed
051	MCID request timed out
052	MCID is not configured for this interface
053	Unknown error
054	Initialization error

Feature Type

Feature type is reported in **ft**_*xxx* format:

Value for *xxx*	Description
001	Fax
002	Modem
003	Modem_phase
004	Hookflash
005	OnHook
006	OffHook

Leg Setup Status

Leg setup status is reported in **ls**_*xxx* format:

Value for *xxx*	Description
000	The call is active or was successful.
001	The outgoing call leg was looped.

Value for *xxx*	Description
002	The call setup timed out (meaning that the destination phone was alerting, but no one answered). The limit of this timeout can be specified in the **leg setup** command.
003	The call setup failed because of a lack of resources in the network.
004	The call setup failed because of an invalid number.
005	The call setup failed for reasons other than a lack of resources or an invalid number.
006	Unused; setup failure.
007	The destination was busy.
008	The incoming side of the call disconnected.
009	The outgoing side of the call disconnected.
010	The conferencing or connecting of the two call legs failed.
011	Supplementary services internal failure
012	Supplementary services failure
013	Supplementary services failure. Inbound call leg was disconnected.
014	The call was handed off to another application.
015	The call setup was terminated by an application request.
016	The outgoing called number was blocked.
026	Leg redirected
031	Transfer request acknowledge
032	Transfer target alerting (future SIP use)
033	Transfer target trying (future SIP use)
040	Transfer success
041	Transfer success with transfer-to party connected (SIP only)
042	Transfer success unacknowledged (SIP only)
050	Transfer fail
051	Transfer failed, bad request (SIP only)
052	Transfer failed, destination busy
053	Transfer failed, request cancelled
054	Transfer failed, internal error
055	Transfer failed, not implemented (SIP only)
056	Transfer failed, service unavailable or unsupported
057	Transfer failed, leg disconnected
058	Transfer failed, multiple choices (SIP only)
059	Transfer failed, timeout; no response to transfer request

Media Status

Media status is reported in **ms**_*xyy* format:

***x* indicates the command**		***yy* indicates the status of the command**	
Value for *x*	**Description**	**Value for *yy***	**Description**
0	Status for a **media play** command.	00	The command was successful and the prompt finished.[1]
1	Status for a media record command.	01	Failure
2	Status for a **media stop** command.	02	Unsupported feature or request
3	Status for a **media pause** command.	03	Invalid host or URL specified
4	Status for a **media resume** command.	04	Received disconnected
5	Status for a **media seek** command to forward.	05	The prompt was interrupted by a key press.
6	Status for a **media seek** command to rewind.		

1. Valid for the **media play** command only, because media_done events are not received for successful completion of other media commands.

Subscribe/Notify

The following return codes are defined for Subscribe/Notify events:

- ev_subscribe_done **su**_*xxx*
- ev_notify_done **no**_*xxx*
- ev_unsubscribe_done **us**_*xxx*
- ev_unsubscribe_ind **ui**_*xxx*

where *xxx* in the strings above represent the following:

Value for *xxx*	**Description**
000	Success
001	Pending
002	Generic failure
003	Subscription expired
004	Socket error
005	DNS error
006	Request timed-out error
007	Connection timed-out error
008	Connection create failed
009	Internal error

Value for *xxx*	Description
010	Response error
099	Undefined

Tone Detect

Tone detect is reported in **td**_*xxx* format:

Value for *xxx*	Description
000	Invalid inband signal
001	FAX_V21
002	FAX_CED
003	FAX_CNG
004	MODEM_2100HZ
005	MODEM_2100HZ_PHASE
006	VOICE_SILENCE
007	CP_TONE

Transfer Status

Transfer status is reported in **ts**_*xxx* format:

Value for *xxx*	Description
000	Generic transfer success
001	Transfer success, transfer-to party is alerting
002	Transfer success, transfer-to party is answered
003	Transfer finished; however, the result of the transfer is not guaranteed
004	Transfer request is accepted
005	Transferee is trying to reach transfer-to party
006	Transfer request is rejected by transferee
007	Invalid transfer number
008	Transfer-to party unreachable
009	Transfer-to party is busy

VoiceXML Dialog Completion Status

VoiceXML dialog completion status is reported in **vd**_*xxx* format:

Value for *xxx*	Description
000	Normal completion because of the <exit> tag or execution reaching the end of the document.
001	Termination because of the default VXML event handling requiring VXML termination.
002	Terminated by the Tcl IVR application.
003	Internal failure.

Sample Scripts

Note The scripts that appear in this section are only examples. They are not necessarily intended to be fully functional. For the latest versions of all sample scripts, go to the Developer Support web site at http://www.cisco.com/go/developersupport

SIP Headers

Passing SIP Headers

The following script prompts for an account number, then makes a call to URL "sip:elmo@sip.tgw.com," where the account number is passed in the SIP header under the header named "AccountInfo." Other static headers, such as Subject, To, From, and Priority, are also passed from the script, either as part of the URL or separately.

```
# prompt_digit_xfer.tcl
# Script Version: 2.0.0
#------------------------------------------------------------------
#
# Copyright (c) 2004 by cisco Systems, Inc.
# All rights reserved.
#------------------------------------------------------------------

proc init { } {
    global param

    set param(interruptPrompt) true
    set param(abortKey) *
    set param(terminationKey) #

}

proc act_Setup { } {

    leg setupack leg_incoming
    leg proceeding leg_incoming
    leg connect leg_incoming

    media play leg_incoming http://townsend.cisco.com/vxml/audio/enterAccount.au
}
```

```
proc act_MediaDone { } {
    global param

    set pattern(account) .+
    set callinfo(alertTime) 30
    leg collectdigits leg_incoming param pattern

}
proc act_GotAccount { } {
    global dest
    global callinfo
    global headers
    global account

    set status [infotag get evt_status]

    if {  $status == "cd_005" } {
    set account [infotag get evt_dcdigits]

    # These are additional headers for the setup request message
    set headers(AccountInfo) "[set account]"
    # this Subject header overwrites the one overloaded in the destination URL
    set headers(Subject) "HelloSipTCL"
    set headers(To) "sip:oscar@abc.com"
    set headers(From) "sip:nobody"
    set callinfo(protoHeaders) headers

    # this destination URL has an overloaded header named "Subject"
    set dest "sip:elmo@sip.tgw.com?Subject=Hello&Priority=Urgent"

    leg setup $dest callinfo leg_incoming

    } else {
    puts "\nCall [infotag get con_all] got event $status collecting destination"
    call close
    }
}

proc act_CallSetupDone { } {

    set status [infotag get evt_status]
    puts "\n STATUS=$status"
    if { $status == "ls_000"} {

        timer start leg_timer 30 leg_incoming
        return
    }
    call close
}

proc act_Cleanup { } {



    set status [infotag get evt_status]
        puts "\n STATUS is $status"

#    puts "\nCAME BACK FROM TRANSFER AND CLOSING CALL"
    call close
}

init
requiredversion 2.0
```

```
#----------------------------------
#   State Machine
#----------------------------------
  set fsm(any_state,ev_disconnected)          "act_Cleanup        same_state"
  set fsm(CALL_INIT,ev_setup_indication)      "act_Setup          MEDIAPLAY"
  set fsm(MEDIAPLAY,ev_media_done)            "act_MediaDone      GETDEST"
  set fsm(GETDEST,ev_collectdigits_done)      "act_GotAccount     PLACECALL"
  set fsm(PLACECALL,ev_setup_done)            "act_CallSetupDone  CALLACTIVE"
  set fsm(CALLACTIVE,ev_leg_timer)            "act_Cleanup        same_state"
  set fsm(CALLACTIVE,ev_disconnected)         "act_Cleanup        CALLDISCONNECT"
  set fsm(CALLDISCONNECT,ev_disconnected)     "act_Cleanup        same_state"
  set fsm(CALLDISCONNECT,ev_media_done)       "act_Cleanup        same_state"
  set fsm(CALLDISCONNECT,ev_disconnect_done)  "act_Cleanup        same_state"
  set fsm(CALLDISCONNECT,ev_leg_timer)        "act_Cleanup        same_state"

  fsm define fsm CALL_INIT
```

Retrieving SIP Headers

The following script plays the prompt "The number is," along with the account number received from the originating gateway. Other headers received are displayed in debug messages.

```
# prompt_digit_xfer.tcl
# Script Version: 2.0.0
#------------------------------------------------------------------
#
# Copyright (c) 2004 by cisco Systems, Inc.
# All rights reserved.
#------------------------------------------------------------------

proc init { } {
    global param

    set param(interruptPrompt) true
    set param(abortKey) *
    set param(terminationKey) #

}

proc act_Setup { } {
    leg setupack leg_incoming
    leg proceeding leg_incoming
    leg connect leg_incoming
    infotag set med_language 1

    set ani [infotag get leg_ani]
    set dnis [infotag get leg_dnis]

    puts "\n get_headers.tcl: ani is $ani"
    puts "\n get_headers.tcl: dnis is $dnis"

    set  Subject [infotag get evt_proto_headers Subject]
    puts "\n get_headers.tcl: Subject =  $Subject"

    set  Priority [infotag get evt_proto_headers Priority]
    puts "\n get_headers.tcl: Priority =  $Priority"

    set  From [infotag get evt_proto_headers From]
    puts "\n get_headers.tcl: From =  $From"
```

```
    set  To [infotag get evt_proto_headers To]
    puts "\n get_headers.tcl: To =  $To"

    set  Via [infotag get evt_proto_headers Via]
    puts "\n get_headers.tcl: Via =  $Via"

    set  tsp [infotag get evt_proto_headers tsp]
    puts "\n get_headers.tcl: tsp =  $tsp"

    set  phone_context [infotag get evt_proto_headers phone-context]
    puts "\n get_headers.tcl: phone-context =  $phone_context"

    set  AccountInfo [infotag get leg_proto_headers AccountInfo]
    puts "\n get_headers.tcl - act_Setup: AccountInfo =  $AccountInfo"

    media play leg_incoming tftp://townsend.cisco.com/audio/num_is.au %n$AccountInfo
}

proc act_CallSetupDone { } {

    set status [infotag get evt_status]
    puts "\n STATUS=$status"
    if { $status == "ls_000"} {

        timer start leg_timer 30 leg_incoming
        return
    }
    call close
}

proc act_Cleanup { } {

    set status [infotag get evt_status]
        puts "\n STATUS is $status"

#    puts "\nCAME BACK FROM TRANSFER AND CLOSING CALL"
    call close
}

init
requiredversion 2.0

#-----------------------------------
#   State Machine
#-----------------------------------
  set fsm(any_state,ev_disconnected)         "act_Cleanup        same_state"
  set fsm(CALL_INIT,ev_setup_indication)     "act_Setup          MEDIAPLAY"
  set fsm(PLACECALL,ev_setup_done)           "act_CallSetupDone  CALLACTIVE"
  set fsm(CALLACTIVE,ev_leg_timer)           "act_Cleanup        same_state"
  set fsm(CALLACTIVE,ev_disconnected)        "act_Cleanup        CALLDISCONNECT"
  set fsm(CALLDISCONNECT,ev_disconnected)    "act_Cleanup        same_state"
  set fsm(CALLDISCONNECT,ev_media_done)      "act_Cleanup        same_state"
  set fsm(CALLDISCONNECT,ev_disconnect_done) "act_Cleanup        same_state"
  set fsm(CALLDISCONNECT,ev_leg_timer)       "act_Cleanup        same_state"

  fsm define fsm CALL_INIT
```

Services

The following script demonstrates how to use the service register command and the start service configuration. The script registers as a service to provide data to other scripts and wakes up on a timer to get the latest data.

Service Register and Start

```
#------------------------------------------------------------------
# sample_service.tcl -
# This script will not accept incoming calls.
#------------------------------------------------------------------

set myname "sample_service.tcl"

#------------
# Someone is telling us to terminate. The service will unregister
# when we close.
#
proc act_Terminate { } {
    global myname

    puts "$myname received a terminate event, closing up."
    call close
}

#------------
# Got an initial event when the script starts running. Register as
# as service, get the initial data, and wait on a timer.
#
proc act_Session { } {
    global myname

    puts "$myname is starting up."
    puts "Starting timer for 30 seconds"
    timer start call_timer0 30

    init_data

    set r [service register data_service]
    puts "  Register of data_service returned $r"
}

#-----------
# Simulate getting some data. We could subscribe, or be collecting info
# from calls. Just keep a counter.
#
proc init_data { } {
    global data

    set data(run_time) 0
    set data(call_count) 0
}

proc get_data { } {
    global data

    set data(run_time) [expr $data(run_time)+30]
}
```

```
#------------
# The initial call setup has come in. Output some display to the console.
# If we are the second session, send a message to the first.
#
proc act_Setup { } {
    global myname

    puts "$myname got a setup. Refusing the call."
    leg disconnect leg_all
}

#------------
# We received a message from another session.
#
proc act_Rx_Msg { } {
    global myname
    global data

    set src [infotag get evt_msg_source]
    infotag get evt_msg msg_array
    incr data(call_count)

    puts "$myname got a message. Respond with the data."
    set r [sendmsg $src -p data]
    puts "Send of data to $src returned $r"
}

#------------
# The timer has gone off. Get the data, and reset the timer.
#
proc act_Timer { } {
    puts "Timer went off. Get data, and reset the timer.\n"

    get_data
    timer start call_timer0 30
}

#------------
# Ignore this event. Output the event name to the console.
#
proc act_Ignore { } {
    global myname

    set ev [infotag get evt_event]
    puts "$myname is ignoring event $ev"
}


#---------------------------------
#   State Machine
#---------------------------------
#
  set fsm(any_state,ev_session_indication) "act_Session     same_state"
  set fsm(any_state,ev_session_terminate)  "act_Terminate   same_state"
  set fsm(CALL_INIT,ev_setup_indication)   "act_Setup       same_state"
  set fsm(any_state,ev_msg_indication)     "act_Rx_Msg      same_state"
  set fsm(any_state,ev_call_timer0)        "act_Timer       same_state"
  set fsm(any_state,ev_any_event)          "act_Ignore      same_state"

  fsm define fsm start_state
```

Session Interaction

The following script demonstrates session interaction. The script runs a VoiceXML script to get the list of handles.

```
#
# Global Data Structures
#    handles - list of handles of other sessions (not our own)
#              If they crash, we will not know, so may be out of date.
#    my_handle - handle of this session
#    my_leg - leg id of the initial leg we will keep
#    recorded_names - array of ptrs to RAM based recordings, indexed by handles.
#                     includes our own.
#

#------------
# The init routine runs once when the script is loaded.
# Setup the global media_parm array for use every time we turn on
# digit collect.
#
proc init { } {
    # Code here runs when the script is loaded.

    global media_parm

    puts "
              Loading si_demo_main.tcl

         "
    #
    # Setup parameter array for collecting digits
    #
    set media_parm(interruptPrompt) true
    set media_parm(enableReporting) true
    set media_parm(maxDigits) 1
}

#------------
#
# When the call comes in, just check in with the si demo
# server by firing up the vxml doc that will submit our handle.
#
# I expect it to return a ptr to a recorded name of this caller,
# and a list of active participants in the demo.
#
proc act_Setup { } {
    global my_handle
    global handles
    global my_leg

    set handles ""
    set my_handle [infotag get mod_handle]
    set my_leg [infotag get leg_incoming]
    puts "
         si_demo_main.tcl got a setup for leg $my_leg
             my_handle=$my_handle
         "

    leg setupack leg_incoming
    leg connect leg_incoming

    set parray(handle) [infotag get mod_handle]
    leg vxmldialog leg_incoming -u http://px1-sun/dramstha/si_demo.vxml -p parray
```

```
}

#------------
# Playout the list of participants so far.
#
proc play_list { } {
    global recorded_names

    puts " Playout the list of callers."

    foreach handle [array names recorded_names] {
    lappend playlist $recorded_names($handle)
    }
    if { [llength $playlist] <= 0 } {
    media play leg_incoming http://px1-sun/dramstha/no_others.au
    } else {
    media play leg_incoming http://px1-sun/dramstha/participants_are.au $playlist
    }
}

#------------
#
# Handoff the call to another leg.
#   We could setup a menu to select which one, but here we simply
#   find the first that is not us.
#
proc act_do_handoff { } {
    global handles
    global my_handle

    if { [llength $handles] <= 0 } {
        puts "Weird handofff, now no handles"
    media play leg_incoming http://px1-sun/dramstha/no_others.au
    fsm setstate RUNNING
    return
    }

    foreach handle $handles {
    set ret [handoff callappl leg_incoming $handle -s "Here is a leg"]
    if { $ret != "unavailable" } break;
    }
    if { $ret == "unavailable" } {
    puts "Bailing out on the handoff"
    media play leg_incoming http://px1-sun/dramstha/no_others.au
    fsm setstate RUNNING
    }
    puts "
       $my_handle handed off leg to $handle  status=$ret"
}

#------------
# We got a digit while conferenced. Don't care who sent it, disconnect.
#
proc act_Control_Conf { } {
    puts "Destroying the connection"
    connection destroy con_all
}

#------------
# Main routine to handle a digit when running the demo.
#   Implement the 4 options, playout help for anything else.
#
proc act_Control { } {
    global media_parm
```

```
    global handles

    set digit [infotag get evt_digit]
    puts "Digit entered=$digit"
    leg collectdigits leg_incoming media_parm

    if { [infotag get evt_state_current] != "RUNNING" } {
    puts "
        Ignoring digit when not running yet"
    return
    }

    switch $digit {
    1 {
            play_list
    }

    2 {
        if { [llength $handles] <= 0 } {
        media play leg_incoming http://px1-sun/dramstha/no_others.au
        } else {
        media play leg_incoming http://px1-sun/dramstha/sending_msg.au
        set oparms(id) msg
        foreach handle $handles {
            set r [sendmsg $handle -p oparms]
        }
        }
    }

    3 {
        if { [llength $handles] <= 0 } {
        media play leg_incoming http://px1-sun/dramstha/no_others.au
        } else {
                puts "Playing handing_off.au"
        media play leg_incoming http://px1-sun/dramstha/handing_off.au
        fsm setstate PRE_HANDOFF
        }
    }

    4 {
        media play leg_incoming http://px1-sun/dramstha/si_feature_description.au
http://px1-sun/dramstha/si_demo_description.au
    }

    default {
        media play leg_incoming http://px1-sun/dramstha/si_help.au
    }
    }
}

#------------
# Got a return from the initial submit to the si demo server.
#
# Save the recorded name of this caller, and the list of handles of other
# callers.
# Checkin with the other callers if there are any.
# Now we are up and running, although we may not have everyones
# name until we get responses from the checkin.
#
proc act_submit_done { } {
    global my_name
    global handles
    global media_parm
    global recorded_names
```

```
    global my_handle

    puts "Tcl script si_demo_main got a return from VXML"
    set my_name http://px1-sun/dramstha/si_unknown_caller.au
    set handles ""

    set r_event [infotag get evt_vxmlevent]
    puts "
          Tcl got on return from initial submit:
             evt_vxmlevent=$r_event
          "

    if { [string match error $r_event] == 1 } {
          return
    }
    set rlist [infotag get evt_vxmlevent_params parray]
    set msg "     Returned data from si server includes:"
    foreach i $rlist {
        append msg "\n            $i=$parray($i)\n"
    if { $i == "handles" } {
        set handles $parray($i)
        regsub $my_handle $handles "" $handles
        set index [lsearch $handles $my_handle]
    }
    if { $i == "recorded_name" } {
        set my_name $parray($i)
    }
    }
    puts "
        Returned data from si server includes: $msg"

    set recorded_names([infotag get mod_handle]) $my_name
    checkin
    #
    # We are up and running:
    puts "****RUNNING handles=$handles"
    leg collectdigits leg_incoming media_parm
    media play leg_incoming http://px1-sun/dramstha/si_help.au
}

#------------
# Checkin with other active calls. Send each a checking message.
#
proc checkin { } {
    global handles
    global my_name

    set oparms(recorded_name) $my_name
    set oparms(id) checkin
    foreach handle $handles {
        if { $handle == [infotag get mod_handle] } continue
    set r [sendmsg $handle -p oparms]
    puts "
            Just sent a checkin msg to $handle.
            Status=$r"
    }
}

#------------
# All done, disconnect the calls we are handling.
#
proc act_Cleanup { } {
    set ev [infotag get evt_event]
    puts "Script for callids <[infotag get leg_all]> got event $ev"
```

```
    puts "Closing now."
    call close
}

#------------
#
# Send a submit to remove the incoming leg from the database. Tell the other
# sessions (calls) we are gone.
#
proc act_db_remove { } {
    global handles

    puts "
         Tcl Firing up VXML to submit to remove handle [infotag get mod_handle]"
    set parray(handle) [infotag get mod_handle]
    leg vxmldialog leg_incoming -u http://px1-sun/dramstha/si_demo_disconnect.vxml -p
parray

    set oparms(id) "uncheckin"
    foreach handle $handles {
    set r [sendmsg $handle -p oparms]
    puts "Uncheckin message Send to $handle returned $r"
    }
}

#------------
#
# Got a message from another instance. Handle it:
#   If they are checking in, save the info and respond.
#   If they are responding to a checkin, save the info.
#   If their caller is sending a msg, tell our caller
#
proc act_rx_msg { } {
    global handles
    global recorded_names
    global my_name

    set src [infotag get evt_msg_source]
    set parm_list [infotag get evt_msg parm_array]

    set msg ""
    foreach i $parm_list {
        append msg "\n         parm($i)=$parm_array($i)"
    }
    puts "
      Leg [infotag get leg_incoming] got a message with [llength $parm_list] args
           from $src
           $msg"

    if { [info exists parm_array(id)] } {
    set id $parm_array(id)
    } else {
    set id "unknown"
    }

    if { [info exists parm_array(recorded_name)] } {
    set recorded_name $parm_array(recorded_name)
    } else {
    set recorded_name "http://px1-sun/dramstha/si_unknown_caller.au"
    }

    if { $id == "checkin" } {
        lappend handles $src
    set recorded_names($src) $recorded_name
```

```
        puts "\n  $src checked in. Now handles=
            $handles"
        set oparms(id) "checkin_rsp"
    set oparms(recorded_name) $my_name
        set r [sendmsg $src -p oparms]
        puts "Message Send to $src returned $r"
    }

    if { $id == "checkin_rsp" } {
        puts "\nSomeone responded to our check in"
    if { [lsearch $handles $src] == -1 } {
        puts "
               Error, We don't have src of the checkin response
                  src=$src
              handles=$handles"
    } else {
        set recorded_names($src) $recorded_name
    }
    }

    if { $id == "uncheckin" } {
    set index [lsearch handles $src]
    if { $index == -1 } {
        puts "
               Error, We don't know who is unchecking in
                  src=$src
              handles=$handles"
    } else {
        lreplace $handles $index $index
    }
    puts "
             After uncheckin, handles=$handles"
    }

    if { $id == "msg" } {
        puts "
          Telling our incoming leg we got a message"
        media stop leg_incoming
        media play leg_incoming http://px1-sun/dramstha/received_msg_from.au
$recorded_names($src)
    }
}

#------------
#
# Refuse a handoff of another leg by simply returning it.
#
proc act_fail_handoff { } {
    puts "
        Got a handoff in state [infotag get evt_state_current]. Returning it."
    set legs [infotag get leg_all]
    set leg2 [lindex $legs 1]
    handoff return $leg2 refused
}

#------------
#
# We got a handoff of another leg. Conference him
# in.
#
proc act_handoff { } {
    global leg1
    global leg2
    global my_leg
```

```
    puts "Got a handoff. string=[infotag get evt_handoff_string]"


    set legs [infotag get leg_all]
    set leg1 [lindex $legs 0]
    set leg2 [lindex $legs 1]

    if { [infotag get evt_state_current] != "RUNNING" } {
    puts "
        Returning handoff leg in state [infotag get evt_state_current]"
    handoff return $leg2 success
    fsm setstate RUNNING
    return
    }

    media stop $my_leg
    puts "Connecting legs $leg1 and $leg2 together"
    connection create $leg1 $leg2
}

#------------
#
# The conference create completed. Simply turn on collect digits so
# both callers can disconnect.
#
proc act_create_done { } {
    global leg1
    global leg2
    global media_parm

    puts "Connection Created."
    leg collectdigits $leg1 media_parm
    leg collectdigits $leg2 media_parm
}

#------------
#
# The conference has been destroyed. Return the other leg to it's
# original session.
#
proc act_destroy_done { } {
    global my_leg
    global leg1
    global leg2

    # Figure out which one to keep.
    if { $leg1 == $my_leg } {
    set retleg $leg2
    } else {
    set retleg $leg1
    }

    puts "
      connection destroyed.
         Returning leg $retleg, keeping $my_leg"
    handoff return $retleg "success"
    media play $my_leg http://px1-sun/dramstha/si_help.au
}

#------------
# We got the call leg returned to us after handoff.
# Just go back to running.
#
```

```
proc act_return { } {
    global media_parm

    puts "
        Leg was returned. Going back to running"
    media play leg_incoming http://px1-sun/dramstha/si_help.au
    leg collectdigits leg_incoming media_parm
}

#------------
# Ignore an event.
#
proc act_Ignore { } {
    set ev [infotag get evt_event]
    puts "
     Script for callids <[infotag get leg_all]>
       in state [infotag get evt_state_current]
       is ignoring event $ev"
}

init

#----------------------------------
#   State Machine
#----------------------------------
#
# States:
#    SUBMIT - We got a setup message, and submitted our handle to the
#             server.
#    RUNNING - Waiting for a command from the caller. Could be playing a
#              prompt.
#    HANDED_OFF - We handed off the incoming leg to another session. We
#                 have no legs here.
#    PRE_HANDOFF - playing a prompt to caller telling them we will handoff
#    CONFERENCED - Either setting up (waiting for conf create done) or
#                  actually conferenced.
#    UNCONFING   - Disconnecting the conference prior to returning a leg
#    DBREMOVE - Got a disconnect, removing ourselves from the server Data Base before
#               closing up shop.
#    CALLDISCONNECT - All done. Have disconnected the caller.
#    CALL_INIT - Initial state before a call comes in
#

  set fsm(any_state,ev_any_event)      "act_Ignore          same_state"
  set fsm(any_state,ev_disconnected)   "act_db_remove       DBREMOVE"
  set fsm(any_state,ev_msg_indication) "act_rx_msg          same_state"
  set fsm(any_state,ev_handoff)        "act_fail_handoff    same_state"
  set fsm(any_state,ev_create_done)    "act_create_done     same_state"
  set fsm(any_state,ev_destroy_done)   "act_destroy_done    RUNNING"
  set fsm(any_state,ev_digit_end)      "act_Ignore          same_state"
  set fsm(any_state,ev_media_done)     "act_Ignore          same_state"

  set fsm(RUNNING,ev_handoff)          "act_handoff         CONFERENCED"
  set fsm(CONFERENCED,ev_digit_end)    "act_Control_Conf    UNCONFING"
  set fsm(RUNNING,ev_digit_end)        "act_Control         same_state"
  set fsm(SUBMIT,ev_vxmldialog_done)    "act_submit_done    RUNNING"
  set fsm(DBREMOVE,ev_vxmldialog_done)  "act_Cleanup        CALLDISCONNECT"

  set fsm(CALL_INIT,ev_setup_indication) "act_Setup         SUBMIT"

  set fsm(CALLDISCONNECT,ev_disconnect_done) "act_Ignore    same_state"
  set fsm(PRE_HANDOFF,ev_media_done)     "act_do_handoff    HANDED_OFF"
  set fsm(HANDED_OFF,ev_returned)        "act_return        RUNNING"
  fsm define fsm CALL_INIT
```

Hybrid Scripting

The following script adds the message waiting indication(MWI) feature to an existing Tcl-VoiceXML (VXML) hybrid application that places an outbound call. The MWI feature is implemented using the SIP SUBSCRIBE/NOTIFY mechanism.

The script demonstrates how to separate the VXML portion to handle the IVR and the Tcl portion to handle call control. The VXML dialogs will play prompts and collect inputs as well as interfaces with the HTTP web server. The main Tcl script handles the call control portion which provides the equivalent function of the standard VXML Transfer. When the user calls into the gateway, a VXML dialog is invoked. The VXML dialog queries the backend server with the ANI of the caller. The VXML document returned presents the user with choices of who he can call. This can be the personal phone book or a company directory. In addition, the VXML Transfer element attributes are passed from the VXML document to the Tcl procedure to place call. The returned information also includes a URL which contains the location of VXML dialog to run after the called party hangs up. This VXML dialog emulates a voicemail message which is played to the user if he chooses to listen to them.

The user also hear a prompt indicating new voicemails in his mailbox after the call has been connected. He can choose to listen to the voicemail or ignore it. If he selects to listen to his voicemail, the music on hold treatment will be applied to the called party This feature is enabled when the Tcl application subscribes to a MWI event package running on a SIP server. For this sample, the MWI event package is another Tcl application running on another gateway. The server application will notify the client application when a new message is delivered to the mailbox.

```
#-------------------------------------------------------------------------------
# Procedure init
#
# The init procedure defines the initial parameters of digit collection. Users are
# expected to enter a single DTMF key within 5 secs and are allowed to enter a digit
# before a prompt is played out. This is used when a prompt is played to the user
# indicating a new voicemail and the user may enter any DTMF key to listen to the
# voicemail message.

proc init { } {

    global param

    set param(maxDigits) 1
    set param(interruptPrompt) true
    set param(initialDigitTimeout) 5
}

#-------------------------------------------------------------------------------
# Procedure init_Config
#
# In this procedure, CLI configurations are read
# Users can configure the location of the Tcl script on the router and this is stored in
# the variable, baseURI.
# Users can configure the number of notifications that are sent before the subscription
# closes. The default is 4.
#

proc init_ConfigVars {} {

    global notNum
    global baseURI
```

```
   if {[infotag get cfg_avpair_exists base-url]} {
        set baseURI [string trim [infotag get cfg_avpair base-url]]
   } else {
        set baseURI tftp://TFTP-SERVER/scripts/
   }

   if {[infotag get cfg_avpair_exists notification-num]} {
        set notNum [string trim [infotag get cfg_avpair notification-num]]
   } else {
        set notNum 4
   }

    puts "\n\n\n**** Base Url is $baseURI\n\n"
    puts "\n\n\n**** Number of notifications in CLI is $notNum\n\n"
}


#-------------------------------------------------------------------------------------------
# Procedure init_perCallVars
#
# In this procedure, the global variables are initialized
# The IP address for the SIP server is configured on the router as follows :
#  ip host x.x.x.x sip-server

proc init_perCallVars { } {

    global ani
    global uri
    global menu
    global name
    global mNew
    global subId
    global mTotal
    global bridge
    global subUrl
    global subInfo
    global counter
    global tclProc
    global maxtime
    global mWaiting
    global callActive
    global longpound
    global destination
    global message_status
    global message_summary
    global connectiontimeout

    set uri ""
    set name ""
    set bridge ""
    set baseURI ""
    set tclProc ""
    set maxtime ""
    set longpound ""
    set destination ""
    set connectiontimeout ""

    set counter 0
    set callActive 0
    set subUrl sip:mwi@sip-server

}

#-------------------------------------------------------------------------------------------
```

```
# Procedure act_Start
#
# The act_Start procedure is executed when it receives an ev_setup indication event.
# A setup acknowledgement, call proceeding and connect message is sent to the incoming
# call leg. The parameters in the subscribe request are defined and a subscribe request
# to the MWI event package is sent.
#

proc act_Start { } {

    global ani
    global subId
    global subUrl
    global subInfo
    global baseURI

    init_perCallVars
    infotag set med_language 1

    leg setupack leg_incoming
    leg proceeding leg_incoming
    leg connect leg_incoming

    set ani [infotag get leg_ani]

    set subInfo(event) mwi
    set subInfo(notificationReceiver) mwiClient
    # expiration time is 1 hour
    set subInfo(expirationTime) 3600
    set subinfo(subscriptionContext) context
    set headers(Account) $ani
    set headers(Subject) "Message Waiting Indication Subscription"
    set subInfo(protoHeaders) headers
    set subInfo(protoContents) "This is a subscription request from mwiclient"
    set subInfo(protoContentTypes) "text/plain"

    set subId [subscription open $subUrl subInfo]

}

#-------------------------------------------------------------------------------------
# Procedure act_VxmlDialog1
#
# This procedure is called from act_Notify. It is called once in the application after the
# script receives a ev_notify_ind event.
# In this procedure, the ANI of the user is submitted to sipWebQuery.php script on the
# backend web server. The result is a dynamically generated VXML document which contains
# information required for call setup. The user is prompted to enter the destination
# number of the person he wishes to call. The destination number together with the
# attributes to the VXML transfer element and the URL for the subsequent VXML dialog are
# passed back to the Tcl script. The IP address of the Web server is configured on the
# router as follows: ip host x.x.x.x HTTP-SERVER

proc act_VxmlDialog1 { } {

    global baseURI

    puts "\n\n\n******** Procedure  VxmlDialog1  ********\n\n"

    set vxmlDialog1 {
        <vxml version="2.0">
            <form id="main">
                <catch event="error.badfetch.com">
                    <log>Web Server down ! Submit action in VxmlDialog1 failed. </log>
```

```
                    <exit/>
                </catch>

                <var name="WEB_SERVER" expr="'http://HTTP-SERVER/vxml/'"/>
                <var name="ANI" expr="session.telephone.ani"/>

                <block>
                <submit expr="WEB_SERVER+'sipWebQuery.php'" method="get" namelist="ANI"/>
                </block>

            </form>
        </vxml> }

    leg vxmldialog leg_incoming -u $baseURI -v $vxmlDialog1
    fsm setstate WEBQUERY
}

#-------------------------------------------------------------------------------------------
# Procedure act_SubscribeDone
#
# This procedure is executed when an ev_unsubscribe_done event is received. The status of
# the event is checked. If the subscription failed, the subscription is closed.

proc act_SubscribeDone { } {


    global subId

    puts "\n\n\n******** Procedure  SubscribeDone  ********\n\n"

    set subId [infotag get evt_subscription_id]

    puts "\n\n\n**** Subscription ID is $subId"
    set status [infotag get evt_status]

    switch $status {
        "sn_000" {
        puts "\n\n\n**** Subscription status, $status swith ID: $subId is successful\n\n"
                    }
        "sn_001" {
        puts "\n\n\n**** Subscription status, $status with ID: $subId is pending\n\n"
                    }
        "sn_002" {
        puts "\n\n\n**** Subscription statu, $status swith ID: $subId failed\n\n"
        subscription close $subId
                    }
    }
}

#-------------------------------------------------------------------------------------------
# Procedure act_Notify
#
# This procedure is received when an ev_notify_ind event is received. Both standard and
# non standard SIP headers sent by the server are read. The non standard headers are
# MessageStatus and MessageSummary. MessageStatus indicates that there are new
# voicemail and MessageSummary indicates the number of new voicemails.
# An acknowledgement is sent back to the server in response to the notification event.
# This procedure is called every time an ev_notify_ind event is received (except for the
# last one). The subscription is terminated depending of the value configured
# for the number of times a notification is sent.

proc act_Notify { } {

    global ani
```

```
global mNew
global mTotal
global notNum
global counter
global mWaiting

puts "\n\n\n******** Procedure Notify  ********\n\n"


set status [infotag get evt_status]
puts "\n\n\n**** Status of subscription is $status\n\n"

set subId [infotag get evt_subscription_id]
puts "\n\n\n**** Subscription ID is subId=$subId"

set event_header [infotag get evt_proto_headers Event]
puts "\n\n\n****  Event header is $event_header\n\n"

set hello [infotag get evt_proto_headers "Hello"]
puts "\n\n\n**** Hello Header from Notification is $hello\n\n"

set message_status [infotag get evt_proto_headers "MessageStatus"]
regexp {Messages-waiting:([a-z]+)} $message_status dummy mWaiting
puts "\n\n\n**** Message Waiting is $mWaiting\n\n"

set message_summary [infotag get evt_proto_headers "MessageSummary"]
regexp {Voicemail:([0-9]+)} $message_summary dummy mNew
puts "\n\n\n**** New message is $mNew\n\n"

set content_type [infotag get evt_proto_content_type]
puts "\n\n\n**** Notification content_type=$content_type\n\n"

set content [infotag get evt_proto_content]
puts "\n\n\n**** Notification content=$content\n\n"

set exp_time [infotag get evt_proto_headers Expires]
puts "\n\n\n**** Subscription Expires: $exp_time\n\n"

set notifyRecr [infotag get subscription_info $subId notificationReceiver]
puts "\n\n\n**** Notification Receiver is  $notifyRecr\n\n"


# subscription expired

if {$exp_time == 0} {

    puts "\n\n\n**** Subscription is terminated"

    set headers(Title) "This is the last acknowledgement from the client"
    set ackInfo(protoContents) "Ending subscription"
} else {
    set headers(Title) "This is an acknowledgement from the client"
    set ackInfo(protoContents) "This is CONTENT from client"
}

set headers(Account) $ani
set ackInfo(protoHeaders) headers
set ackInfo(respCode) ack
set ackInfo(protoContentTypes) "text/plain"

subscription notify_ack $subId -i ackInfo

if {$counter == 0} {
    puts "\n\n\n**** Start VXML Dialog \n\n"
```

```
            act_VxmlDialog1
            incr counter
            return
        }

        if {$counter < $notNum} {
              if {$mWaiting == "yes"} {
                   act_MessageIndication
                   incr counter
               }
              return
        }

        puts "\n\n\n**** Number of notifications received is $counter"

        # close the subscription
        puts "\n After $counter notifications, close the subscription now.\n\n"

        subscription close $subId
        fsm setstate CLOSESUB
    }

    #-------------------------------------------------------------------------------------------
    # Procedure act_MessageIndication
    #
    # This procedure is first invoked when the script receives an ev_leg_timer event. The
    # timer event comes up if there is a new voicemail after the call has been connected.
    # To play a prompt to the user that a new voicemail is in the mailbox, the connection
    # between the incoming and outgoing call legs is destroyed.
    #

    proc act_MessageIndication {} {

        global mNew
        global param
        global mTotal
        global mWaiting
        global callActive

        puts "\n\n\n******** Procedure MessageIndication ********\n\n"

        if {$callActive == 1} {
            set callActive 0
            connection destroy con_all
        } else {
            leg collectdigits leg_incoming param

            media play leg_incoming _new_voicemail.au %n$mNew %s1000 _check_new_voicemail.au
        }
        fsm setstate CHECKVOICEMAIL
    }

    #-------------------------------------------------------------------------------------------
    # Procedure act_PlayMessageInd
    #
    # This procedure plays the prompt "You have X new voicemails" to the user when the script
    # receives an ev_destroy_done event is received. The act_PlayMusic is invoked to provide
    # music on hold treatment to the called party

    proc act_PlayMessageInd {} {

        global mNew
        global param
        global mTotal
```

```
    global mWaiting

    puts "\n\n******* Procedure PlayMessageInd ********\n\n"
    puts "\n\n\n**** Number of new vmail are $mNew\n\n"

    leg collectdigits leg_incoming param
    media play leg_incoming _new_voicemail.au  %n$mNew %s1000 _check_new_voicemail.au
        #media play leg_incoming _you_have.au %n$mNew %s1000 _seconds.au

    act_PlayMusic

    fsm setstate CHECKVOICEMAIL
}

#-------------------------------------------------------------------------------------------
# Procedure act_PlayMusic
#
# This procedure plays a music prompt to the called party while the user is checking
# his voicemail messages.
#
proc act_PlayMusic {} {

    puts "\n\n******* Procedure PlayMusic ********\n\n"

    media play leg_outgoing _song.au
    fsm setstate MUSICONHOLD
}

#-------------------------------------------------------------------------------------------
# Procedure act_FinalNotify
#
# This procedure is invoked when the script receives the final ev_notify_ind event.
# The SIP server application sends this event when the client script closes the
# subscription.

proc act_FinalNotify { } {

    puts "\n\n******* Procedure FinalNotify ********\n\n"

    set status [infotag get evt_status]
    puts "\n\nstatus=$status\n\n"
    set sub_id [infotag get evt_subscription_id]

    # you can access any standard headers
    set From [infotag get evt_proto_headers From]
    puts "\n\n**** From header is: $From\n\n"

    set headers(Title) "Hello, this is the final acknowledgement from the client"
    set ackInfo(protoHeaders) headers
    set ackInfo(protoContents) "This is CONTENT from client"
    set ackInfo(protoContentTypes) "text/plain"
    set ackInfo(respCode) ack

    subscription notify_ack $sub_id -i ackInfo
}

#-------------------------------------------------------------------------------------------
# Procedure act_UnsubscribeDone
#
# This procedure is invoked when the script receives an ev_unsubscribe_done event. This
# event is sent when the subscription is closed.

proc act_UnsubscribeDone { } {
```

```
    puts "\n\n\n******** Procedure UnsubscribeDone  ********\n\n"

    set status [infotag get evt_status]
    puts "\n\n\n**** Status of unsubscribe_done is $status\n\n"

}

#-------------------------------------------------------------------------------------
# Procedure act_SubscribeClose
#
# This procedure is invoked when the script receives an ev_unsubscribe_indication or an
# ev_subscribe_cleanup event. An ev_unsubscribe_indcation event is received if the server
# sends unsubscribe request, subscription  expires or other errors. An
# ev_subscribe_cleanup event is received when the CLI, "clear sub all" is executed
# The subscription is closed.
#

proc act_SubscribeClose { } {

    puts "\n\n\n******** Procedure SubscribeClose  ********\n\n"

    puts "\n\n**** Unsubscribe Indication received or Subscribe Clean up event
    received\n\n"
    set sub_id [infotag get evt_subscription_id]
    subscription close $sub_id
}


#-------------------------------------------------------------------------------------
# Procedure act_GetTransferAttr
#
# This procedure handles the transfer attributes sent by VxmlDialog1.
# It parses the evParam array to get the attributes.

proc act_GetTransferAttr {} {

    global uri
    global name
    global bridge
    global evParam
    global tclProc
    global maxtime
    global longpound
    global destination
    global transfer_param
    global connectiontimeout

    puts "\n\n\n******** Procedure GetTransferAttr  ********\n\n"

    set tclProc GetTransferAttr
    set ev [infotag get evt_vxmlevent]

    if {$ev != "vxml.dialog.transferEvent"} {
        puts "\n\n\t\t ****  Expected event vxml.dialog.transferEvent, got $ev"
    }

    # get all the parameters sent from VXML dialog

    set eventParam [infotag get evt_vxmlevent_params evParam]

    for {set i 0} {$i < [llength $eventParam]} {incr i} {
        set [set value [lindex $eventParam $i]] $evParam($value)
        puts "\n\n\t\t **** $value = $evParam($value)"
    }
```

```
}

#-------------------------------------------------------------------------------------
# Procedure act_Transfer
#
# This procedure is called when the application receives the vxml_dialog_done event.
# It checks for the destination number, set the callInfo array with the relevant
# transfer attributes and places a call to that number.

proc act_Transfer { } {

    global uri
    global name
    global bridge
    global tclProc
    global maxtime
    global longpound
    global destination
    global transfer_param
    global connectiontimeout

    puts "\n\n\n******** Procedure Transfer ********\n\n"

    set tclProc Transfer

    # check the sub-event name
    set exp_ev vxml.session.complete
    set ev [infotag get evt_vxmlevent]

    if {$ev != $exp_ev} {
        puts "\n\n\t\t **** Expected event $exp_ev, got $ev"
    }

     # check the dialog status
     set status [infotag get evt_status]

     switch $status {
                    "vd_000" {
                              #get the transfer attribute, destination
                              infotag get evt_vxmlevent_params transfer_param

                              if !{[info exists transfer_param(destination)]} {
                                  puts "\n\n\t\t ****  destination number does not exist"
                                  act_LogStatus $status $tclProc
                              }

                              set destination $transfer_param(destination)
                              set callInfo(alerTime) $maxtime
                              set callInfo(newguid) $longpound
                              leg setup $destination callInfo leg_incoming
                    }

                   "vd_001" -
                   "vd_002" -
                   "vd_003" {
                             puts "\n\n\t\t ****  VXML Dialog status, expected vd_000, got
                            $status"
                             act_LogStatus $status $tclProc
                   }
                  }
}

#-------------------------------------------------------------------------------------
# Procedure act_TransferDone
```

```
#
# If leg setup is successful, the 2 parties are conferenced. If not, the status of leg
# setup is sent to the web server. If there are new voicemail messages, a timer is
# started for 2 seconds and act_MessageIndication procedure is called.

proc act_TransferDone { } {

    global mNew
    global mTotal
    global tclProc
    global mWaiting
    global incoming
    global outgoing
    global callActive
    global destination
    global transferStatus

    puts "\n\n\n******** Procedure TransferDone  ********\n\n"

    set tclProc TransferDone
    set status [infotag get evt_status]

    puts "\n\n\t\t **** Status of leg setup is $status \n"
    switch $status {
        "ls_000" {
            set incoming [infotag get leg_incoming]
            set outgoing [infotag get leg_outgoing]
            set  creditTimeLeft [infotag get leg_settlement_time leg_all]

            if { ($creditTimeLeft == "unlimited") || ($creditTimeLeft=="uninitialized")}
                {puts "\n\n\t\t **** UnLimited Call Time\n"
                    set callActive 1
                    puts "\n\n\t\t **** Are there new voicemail messages ? $mWaiting \n\n"
                    if { $mWaiting == "yes" } {timer start leg_timer 2 leg_incoming}}
                    else {
                                # start the timer for ...
                                if { $creditTimeLeft < 10 } {
                                   set beep 1
                                   set delay $creditTimeLeft
                                } else {
                                set delay [expr $creditTimeLeft - 10]
                                }
                                timer start leg_timer $delay leg_incoming
                            }
                  }

            "ls_007" {puts "\n\n\t\t **** Call status is $status, Destination is Busy \n"
                set transferStatus NOANSWER
                act_LogStatus $transferStatus $tclProc
                  }
            "ls_008" {puts "\n\n\t\t **** Call status is $status, Incoming Disconnected
                \n"
                set transferStatus NEAR_END_DISCONNECT
                act_LogStatus $transferStatus $tclProc
                  }
                 "ls_009" {puts "\n\n\t\t **** Call status is $status, Outcoming Disconnect
                \n"
                set transferStatus FAR_END_DISCONNECT
                act_LogStatus $transferStatus $tclProc
                 }
                 default {
                           puts "\n\n\t\t **** Call status is $status\n"
                           set transferStatus UNKNOWN
                           act_LogStatus $transferStatus $tclProc
```

```
                }
    }
}

#-------------------------------------------------------------------------------------
# Procedure act_VxmlDialog2
#
# This procedure is called when the caller chooses to listen to his new voicemail
# messages, the script receives an ev_collectdigits_done event.
# After the called party hangs up, the application invokes the second VXML dialog which
# is a fake voicemail service for the user.

proc act_VxmlDialog2 { } {

    global uri
    global menu
    global baseURI
    global incoming
    global outgoing
    global callActive

    puts "\n\n\n******** Procedure VxmlDialog2  ********\n\n"
    puts "\n\n\t\t **** URL of voicemail is $uri \n\n"

    if {$callActive == 1} {
        leg disconnect leg_outgoing
        set callActive 0
    }

    set event [infotag get evt_event]
    set status [infotag get evt_status]

    switch $status {
                    "cd_005" {
                              puts "\n\n\n**** Caller wants to check new voicemail \n\n"
                              if {$uri != ""} {
                                  leg vxmldialog leg_incoming -u $uri
                              } else {
                                  # play media to say cannot access voicemails
                                  media play leg_incoming _technicalProblem.au
                              }

                    }
                    "cd_001" {
                              puts "\n\n\n**** Caller didn't enter digit to check new
voicemail\n\n"
                              media stop $outgoing

                              if {$callActive == 0} {
                                  connection create $incoming $outgoing
                                  set callActive 1
                                  return
                               }
                    }
                    "cd_010" {
                            puts "\n\n\n**** The digit collection was terminated because
                          of an unsupported or unknown feature or event\n\n"
                            return
                    }
    }
}

#-------------------------------------------------------------------------------------
# Procedure act_VxmlDialog2Done
```

```
#
# This procedure is invoked when the ev_vxmldialog_done event is received.
# The conference between the incoming and outgoing call legs is bridged again after the
user is done listening to the new voicemail messages.

proc act_VxmlDialog2Done { } {

    global tclProc
    global incoming
    global outgoing
    global callActive

    puts "\n\n\n******** Procedure VxmlDialog2Done  ********\n\n"

    set tclProc VxmlDialog2Done
    set exp_ev vxml.session.complete
    set ev [infotag get evt_vxmlevent]

    if {$ev != $exp_ev} {
        puts "\n\n\t\t **** Expected event $exp_ev, got $ev"
        act_LogStatus $ev $tclProc
    }

    if {[infotag get evt_legs] == $incoming} {
        media stop $outgoing
        if {$callActive == 0} {
           connection create $incoming $outgoing
           set callActive 1
           return
        }

   }
}

#-------------------------------------------------------------------------------------------
# Procedure act_LogStatus
#
# The status code for leg setup, vxml dialog are sent to the backend web server in this
# procedure.

proc act_LogStatus {statusCode tclProcedure} {

    global baseURI

    puts "\n\n\n******** Procedure LogStatus  ********\n\n"
    puts "\n\n\t\t***** Status Code is $statusCode in procedure $tclProcedure"

    set vxmlDialog3 {
        <vxml version="2.0">
            <form id="main">

                <var name="WEB_SERVER" expr="'http://HTTP-SERVER/vxml/'"/>
                <var name="ANI" expr="session.telephone.ani"/>
                <var name="STATUSCODE" expr="com.cisco.params.code"/>
                <var name="PROCEDURE" expr="com.cisco.params.procedure"/>

                <catch event="error.badfetch.com">
                    <log>Web Server down ! Submit action in VxmlDialog3 failed. </log>
                    <exit/>
                </catch>

                <block>
                    <log> Tcl Status Code : <value expr="STATUSCODE"/> found in Tcl
                    Procedure : <value expr="PROCEDURE"/></log>
```

```
                    <submit expr="WEB_SERVER+'status.php'" method="get" namelist="ANI
                    STATUSCODE PROCEDURE"/>
                </block>

            </form>
        </vxml> }

    set tclStatusParam(code) $statusCode
    set tclStatusParam(procedure) $tclProcedure

    leg vxmldialog leg_incoming -u $baseURI -v $vxmlDialog3 -p tclStatusParam
    fsm setstate LOGSTATUS

}

#-------------------------------------------------------------------------------------------
# Procedure act_Timer
#
# This procedure is invoked when the timer expires. If the call duration is unlimited,
# this will not be invoked.

proc act_Timer { } {

    global beep
    global incoming
    global outgoing

    puts "\n\n\n******** Procedure Timer  ********\n\n"

    set incoming [infotag get leg_incoming]
    set outgoing [infotag get leg_outgoing]

    if { $beep == 0 } {
        #insert a beep ...to the caller
        connection destroy con_all
        set beep 1
    } else {
        connection destroy con_all
        fsm setstate LASTWARN
    }
}

proc act_LastWarn { } {
    media play leg_incoming _outOfTime.au
}

proc act_Destroy { } {
    media play leg_incoming _beep.au
}

#-------------------------------------------------------------------------------------------
# Procedure act_PlayMessageIndDone
# When the script receives an ev_media_done event is received, this procedure will be
# executed. If the event happened on the outgoing leg, the music prompt is repeated to
# the called party. If the event is received on the incoming leg,  the prompt playout
# is terminated on the outgoing side and the 2 legs are  conferenced.

proc act_PlayMessageIndDone { } {

    global callActive
    global incoming
    global outgoing

    puts "\n\n\n******** Procedure PlayMessageIndDone  ********\n\n"
```

```
    set incoming [infotag get leg_incoming]
    set outgoing [infotag get leg_outgoing]

    if {[infotag get evt_legs] == $outgoing} {
         media play leg_outgoing _song.au
         return
    }
    if {[infotag get evt_legs] == $incoming} {
        media stop $outgoing
        if {$callActive == 0} {
           connection create $incoming $outgoing
           set callActive 1
           return
        }
    }
}
proc act_Beeped { } {

    global incoming
    global outgoing

    connection create $incoming $outgoing
}

proc act_ConnectedAgain { } {

    puts "\n\n\t\t***** Call Connected Again\n\n "
}

#-------------------------------------------------------------------------------------
# Procedure act_HandleOutgoing
#
# When the called party hangs up, the connection is destroyed

proc act_HandleOutgoing { } {

    global outgoingDisconnect

    puts "\n\n\t\t********* Procedure HandleOutgoing *********\n\n "

    if {[infotag get evt_legs] == [infotag get leg_outgoing]} {

        # Outgoing disconnected

        connection destroy con_all
        set outgoingDisconnect 1
    } else {
        call close
        fsm setstate CALLDISCONNECT
    }
}

#-------------------------------------------------------------------------------------
# Procedure act_LongPound
#
# If the user enters the Pound key, the connection to the called party will be destroyed.
# The user hear the voicemail emulation in VxmlDialo2

proc act_LongPound { } {

    puts "\n\n\n******** Procedure LongPound  ********\n\n"
    puts "\n\n\t\t***** Calling Party  entered long pound"
```

```
    set DURATION 1000

    if {[infotag get evt_digit] != "#"} {
        fsm setstate same_state
    } else {
        set duration [infotag get evt_digit_duration]

        if {$duration < $DURATION} {
            fsm setstate same_state
        } else {
            connection destroy con_all
        }
    }
}

#-------------------------------------------------------------------------------------
# Procedure act_Cleanup
#
# This procedure is invoked when the application receives an ev_disconnected event.

proc act_Cleanup { } {

    puts "\n\n\n******** Procedure Cleanup  ********\n\n"
    call close
}

#-------------------------------------------------------------------------------------
# Procedure act_SessionClose
#
# This procedure will be invoked if a script receives an ev_session_terminate event.
# Note that this will not be invoked in this application since this application requires
an incoming call leg

proc act_SessionClose { } {

    puts "\n\n\n******** Procedure SessionClose  ********\n\n"
    call close
}

proc act_Ignore { } {
# Dummy Procedure
    puts "Event Capture"
}


requiredversion 2.0
init

init_ConfigVars

#---------------------------------
#   State Machine
#---------------------------------
  set fsm(any_state,ev_disconnected)             "act_Cleanup               same_state"
  set fsm(any_state,ev_subscribe_done)             "act_SubscribeDone         SUBSCRIBED"
  set fsm(any_state,ev_notify_ind)                 "act_Notify                NOTIFIED"
  set fsm(any_state,ev_unsubscribe_done)           "act_UnsubscribeDone       same_state"
  set fsm(any_state,ev_unsubscribe_indication)     "act_SubscribeClose        same_state"
  set fsm(any_state,ev_subscribe_cleanup)          "act_SubscribeClose        same_state"
  set fsm(any_state,ev_session_terminate)          "act_SessionClose          same_state"

  set fsm(CALL_INIT,ev_setup_indication)           "act_Start                 SUBSCRIBE"
  set fsm(NOTIFIED,ev_media_done)                  "act_Ignore                same_state"
  set fsm(CHECKVOICEMAIL,ev_collectdigits_done)    "act_VxmlDialog2           same_state"
```

```
set fsm(MUSICONHOLD,ev_media_done)              "act_PlayMessageIndDone    same_state"
set fsm(CLOSESUB,ev_notify_ind)                 "act_FinalNotify           same_state"

set fsm(WEBQUERY,ev_vxmldialog_event)           "act_GetTransferAttr       same_state"
set fsm(WEBQUERY,ev_vxmldialog_done)            "act_Transfer              TRANSFER"
set fsm(TRANSFER,ev_setup_done)                 "act_TransferDone          CALLACTIVE"
set fsm(CALLACTIVE,ev_leg_timer)              "act_MessageIndication     CHECKVOICEMAIL"
set fsm(CALLACTIVE,ev_disconnected)           "act_HandleOutgoing        CONNDESTROY"
set fsm(CALLACTIVE,ev_digit_end)              "act_LongPound             CONNDESTROY"
set fsm(CHECKVOICEMAIL,ev_vxmldialog_done)    "act_VxmlDialog2Done       same_state"
set fsm(CHECKVOICEMAIL,ev_create_done)        "act_ConnectedAgain        CALLACTIVE"
set fsm(LOGSTATUS,ev_vxmldialog_done)         "act_Cleanup               same_state"
set fsm(CHECKVOICEMAIL,ev_destroy_done)       "act_PlayMessageInd        same_state"
set fsm(PLAYBEEP,ev_media_done)               "act_Beeped                same_state"
set fsm(INSERTBEEP,ev_destroy_done)           "act_Destroy               same_state"
set fsm(INSERTBEEP,ev_media_done)             "act_Beeped                same_state"
set fsm(INSERTBEEP,ev_create_done)            "act_ConnectedAgain        CALLACTIVE"
set fsm(LASTWARN,ev_destroy_done)             "act_LastWarn              CALLDISCONNECT"
set fsm(CALLDISCONNECT,ev_disconnect_done)    "act_Cleanup               same_state"
set fsm(CALLDISCONNECT,ev_media_done)         "act_Cleanup               same_state"
set fsm(CALLDISCONNECT,ev_disconnect_done)    "act_Cleanup               same_state"
set fsm(CALLDISCONNECT,ev_leg_timer)          "act_Cleanup               same_state"

fsm define fsm CALL_INIT
```

This chapter lists common terms and acronyms used throughout this document. For a more detailed list of internetworking terms and acronyms, refer to the Internetworking and Acronyms web site at:

http://www.cisco.com/univercd/cc/td/doc/cisintwk/ita/index.htm

A

AAA Authentication, authorization, and accounting. A suite of network security services that provides the primary framework through which you can set up access control on your Cisco router or access server.

ANI Automatic number identification. Same as calling party.

API Application programming interface.

AV-pair An attribute-value pair used in authentication.

C

CDR Call data record.

CLI Command-line interface.

connection The tying together of two streams or call legs so that the incoming voice stream of one call leg is sent as the outgoing voice stream of the other call leg.

D

DID Direct inward dial. Calls in which the gateway uses the number that you initially dialed (DNIS) to make the call instead of prompting you to dial additional digits.

DNIS Dialed number information service.

DSP Digital signaling processor.

DTMF Dual tone multi-frequency. Use of two simultaneous voice-band tones for dialing (such as touch tone).

E

execution instance An instance of the Tcl interpreter that is created to execute the script.

F

FSM Finite State Machine.

I

IE Information element.

IVR Interactive voice response. Term used to describe systems that provide information in the form of recorded messages over telephone lines in response to user input in the form of spoken words or, more commonly, DTMF signaling. Examples include banks that allow you to check your balance from any telephone and automated stock quote systems.

R

RADIUS Remote Authentication Dial-In User Service. A protocol used for access control, such as authentication and authorization, or accounting.

RTSP Real-Time Streaming Protocol. Enables the controlled delivery of real-time data, such as audio and video. Sources of data can include both live data feeds, such as live audio and video, and stored content, such as prerecorded events. RTSP is designed to work with established protocols, such as RTP and HTTP.

T

Tcl Tool Command Language. A scripting language used for gateway products both internally and externally to Cisco IOS software code.

TFTP Trivial File Transfer Protocol. Simplified version of FTP that allows files to be transferred from one computer to another over a network, usually without the use of client authentication (for example, username and password).

TTS Real Time Streaming Protocol. Enables the controlled delivery of real-time data, such as audio and video. Sources of data can include both live data feeds, such as live audio and video, and stored content, such as prerecorded events. RTSP is designed to work with established protocols, such as RTP and HTTP.

U

URI Uniform Resource Identifier. Type of formatted identifier that encapsulates the name of an Internet object, and labels it with an identification of the name space, thus producing a member of the universal set of names in registered name spaces and of addresses referring to registered protocols or name spaces. [RFC 1630]

V

VoFR Voice over Frame Relay. VoFR enables a router to carry voice traffic (for example, telephone calls and faxes) over a Frame Relay network. When sending voice traffic over Frame Relay, the voice traffic is segmented and encapsulated for transit across the Frame Relay network using FRF.12 encapsulation.

VoIP Voice over IP. The capability to carry normal telephony-style voice over an IP-based internet with POTS-like functionality, reliability, and voice quality. VoIP enables a router to carry voice traffic (for example, telephone calls and faxes) over an IP network. In VoIP, the DSP segments the voice signal into frames, which then are coupled in groups of two and stored in voice packets. These voice packets are transported using IP in compliance with ITU-T specification H.323.