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 , "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 , "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 , "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 , "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 , "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 , "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 , "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 , "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 , "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 , "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 , "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 legXto param

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

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 , "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 , "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 , "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 -48, 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 , "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 , "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 , "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 , "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 , "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 -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 -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 , "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 , "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 5-14, 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 , "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 , "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 , “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 , "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 , "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 , "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 , "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 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.

- *sourceCarrierID*—Source trunk group carrier ID
- *targetCarrierID*—Target trunk group carrier ID

  These parameters can accept a URL string. These parameters do 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.

– fax—Sets the calling number to send fax message.This parameter does not allow indexing.

– `argstring`— Sets the arg-string option that specifies a string or list of strings to be passed to the destination. This parameter does not allow indexing.

– pinNum—Sets the pin number. 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.
- *sourceCarrierID*—The ID of the source carrier.
- *targetCarrierID*—The ID of the target carrier.

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
set callInfo (sourceCarrierID)
set callInfo (targetCarrierID)
```

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