Text Part Number: OL-0350-01

TCL IVR API Version 1.0 Programmer's Guide

Version Date: 03/11/02

This document contains information about the Tool Command Language (TCL) Interactive Voice Response (IVR) application programming interface (API) Version 1.0 commands that you can use to write TCL scripts to interact with the Cisco IVR feature. It provides an annotated example of a TCL IVR script and includes instructions for testing and loading a TCL IVR script.

This document helps developers writing voice application software for Cisco voice interfaces, such as the Cisco AS5x00 series. This includes independent software vendors (ISVs), in-house corporate developers, system integrators, and Original Equipment Manufacturers (OEMs).

This document includes the following sections:

Overview of IVR and TCL

IVR is a term that is used to describe systems that provide information in the form of recorded messages over telephone lines in response to user input in the form of spoken words, or more commonly dual tone multifrequency (DTMF) signaling. For example, when a user makes a call with a debit card, an IVR application is used to prompt the caller to enter a specific type of information, such as a PIN. After playing the voice prompt, the IVR application collects the predetermined number of touch tones (digit collection), forwards the collected digits to a server for storage and retrieval, and then places the call to the destination phone or system. Call records can be kept and a variety of accounting functions performed.

Corporate Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA

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

The prompts used in an IVR script can be either *static* or *dynamic*. With static prompts, the prompt message is pre-recorded in its entirety and played as a single unit. With dynamic prompts, the prompt message is pre-recorded in pieces and the pieces are assembled dynamically during play. For example, dynamic prompts are used to inform the caller of how much time is left in their debit account, such as:

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

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

The Cisco IVR (Interactive Voice Response) feature, available in Cisco IOS Release 12.0(6)T and later, provides IVR capabilities using TCL 1.0 scripts. These scripts are signature locked, and can be only modified by Cisco. The IVR feature allows IVR scripts to be used during call processing. The scripts interact with the IOS software to perform various call-related functions. Starting with release 12.1(3), the TCL script lock will be removed, thus allowing customers to create and change their own TCL scripts.

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

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

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

Plans for Developing a New Script Format

Cisco plans to develop a new TCL script format (TCL 2.0) that will address scalability and performance issues.

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

Prerequisites

The Open TCL IVR feature is currently supported on the Cisco AS5300 voice platform only. In order to use the open TCL IVR feature you need the following:

1. Cisco AS5300 (Voice platform) universal access server

2. Cisco IOS Release 12.1(3) or later.

3. TCL version 7.1 or later.

Calls can come into a gateway using analog lines, ISDN lines, a VoIP link, or a VoFR link. TCL IVR scripts can provide full functionality for calls received over analog or ISDN lines. The functionality provided for calls received over VoIP or Voice over Frame Relay (VoFR) links varies depending on the release of Cisco IOS software being used. For example, if you are using Cisco IOS Release 12.0, you cannot play prompts or tones, and you cannot collect tones.

Restrictions

- The argument **av-send**, used with the **authenticate** and **authorize** commands, is valid on Cisco IOS Release 12.1(2) and later. Refer to the **authenticate** and **authorize** for more information.
- If a TCL 1.0 script is used with the TCL Interpreter 8.0.5, the minimum memory requirement is 64 mbytes for the Cisco 2600 and Cisco 36xx platforms, starting with Cisco IOS Release 12.1(3)T. Additional memory is only needed if one uses TCL 1.0 scripts with TCL interpreter 8.0.5. At this time, the unlocked TCL scripts are not supported on the 2600 and the 36xx platforms. This memory requirement applies regardless of whether the TCL scripts are locked or unlocked.

Developer Support

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

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

How to Use TCL IVR Scripts

This section of the document includes information on how to create and use TCL IVR scripts. This section includes the following topics:

Writing an IVR Script using TCL Extensions

Before you write an IVR script using TCL, you should familiarize yourself with the TCL extensions for IVR scripts.

You can use any text editor to create your TCL IVR script. Follow the standard conventions for creating a TCL script and incorporate the TCL IVR commands as necessary.

A sample script is provided in this section to illustrate how the TCL IVR commands can be used.

Note When writing scripts, it's strongly recommended first doing a **setupAck** or an **acceptCall**.

Note If the caller hangs up, the script stops running and the call legs are cleared. No further processing will be done by the script, which means the **exit** command will not be called.

Standard TCL Commands Used in TCL IVR Scripts

Certain standard TCL commands can be used in TCL IVR scripts. These commands are as follows:

append	**array**	**break**	**case**
catch	**concat**	**continue**	**error**
eval	**expr**	**for**	**foreach**
format	**global**	**history**	**if**
incr	**info**	**join**	**lappend**
lindex	**linsert**	**list**	**llength**
lrange	**lreplace**	**lsearch**	**lsort**
proc	**puts**	**regexp**	**regsub**
rename	**return**	**set**	**split**
string	**switch**	**tcl_trace**	**unset**
uplevel	**upvar**	**while**	

For additional information about the standard TCL commands, see the *TCL and the TK Toolkit* by John Ousterhout (published by Addison Wesley Longman, Inc).

TCL IVR Commands At-a-Glance

In addition to the standard TCL commands, you can use the TCL extensions for IVR scripts that Cisco has created. Also, Cisco modified two of the existing TCL commands, **puts** and **exit**, to perform specific tasks. The TCL IVR commands are as follows:

Command	Description
acceptCall	Sends setup acknowledgement, call proceeding, and call connect messages to the incoming call leg.
ani	Returns the originating number, which is supplied by the incoming call leg.
authenticate	Sends an authentication request to an external system, typically a Remote Access Dial- In User Services (RADIUS) server.
authorize	Sends an authorization request to an external system, typically a RADIUS server.

callID	Returns the call ID used in the Cisco IOS debug messages.
callProceeding	Sends a call proceeding message to the incoming call leg.
clearOutgoingLeg	Deletes a conference and disconnects the outgoing call leg.
conferenceCreate	Creates a conference between two call legs.
conferenceDelete	Deletes a conference between two call legs.
creditTimeLeft	Returns the credit time available for this call.
did	Determines whether the incoming dial peer is configured for dial-in direct (DID) operation.
dnis	Returns the destination number.
exit	Exits the script, clears the call leg, and cleans up the data structures.
failureCode	Returns the last failure event and code.
getVariable	Returns the value of a specified variable.
insertMessage	Inserts a message to the originating (calling) party.
insertTone	Inserts a beep to the originating (calling) party.
placeCall	Places a call to the specified destination.
playFailureTone	Plays one of two tones depending on the last failure event.
playPrompt	Plays a prompt to the incoming call leg. Designed for use with dynamic prompts.
playTone	Plays a call progress tone.
promptAndCollect	Plays a prompt to the originating (calling) party and collects digits in a specified pattern.
puts	Outputs a debug string to the console if the IVR state debug flag is set.
rdn	Returns the redirect number that came with the call.
setLocation	Sets the location of the audio files.
setTimeout	Sets the initial digit collection timeout and the inter-digit collection timeout values.
setVariable	Sets the value of a specified variable.
setupAck	Sends a call setup acknowledgement back to the incoming call leg.
startTimer	Causes the script to block until a specified number of seconds passes or an event (such as a destination hang up or an origination hang up) occurs.
waitEvent	Causes the script to pause until an event occurs.

For more information about these commands, see the “TCL IVR Command Reference” on page 21.

Sample TCL IVR Script

The following annotated example illustrates how the TCL IVR commands can be used.

We start with the header information, which includes the name of the script, the date the script was created and by whom, and some general information about what the script does.

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

```
# Script Loaded by: Dram
# Script Version 1.0.1
# Script LookDate: Sept. 30 20:58:49 1999
#-----------------------------------------------------------------
# September 1998, David Ramsthaler
#
# Copyright (c) 1998, 1999, 2000 by cisco Systems, Inc.
# All rights reserved.
#-----------------------------------------------------------------
#
# This script authenticates using (ani, dnis) for (account, password). If
# that fails, it collects account and pin number, then authenticates
# using (account, pin).
#
# If authentication passes, it collects the destination number and
# places the call.
#
```

Next, we define a series of procedures.

The **get_account** procedure defines the parameters of the account prompt and collects the account information. In this procedure:

- The prompt plays an audio file called enter_account, which is stored in Flash memory.
- The user is allowed to enter information before the prompt message is complete.
- The user is allowed to abort the process by pressing the asterisk key.
- The user must indicate that they have completed their entry by pressing the pound (#) key.
- The input from the user is collected in an array.
- If the collection of information is successful, the script stores the account number and indicates that the next procedure is to get the personal identification number (PIN).
- If the user aborts the process, the script indicates that this procedure should be started again.
- If the account number is not collected, the script indicates that this procedure should be started again.

```
proc do_get_account {} {
    global state
    global account

    set prompt(url) flash:enter_account.au
    set prompt(interrupt) true
    set prompt(abortKey) *
    set prompt(terminationKey) #
    set patterns(account) .+
    set event [promptAndCollect prompt info patterns ]

    if {$event == "collect success"} {
        set state get_pin
```

```
        set account $info(digits)
        return 0
    }

    if {$event == "collect aborted"} {
        set state get_account
        return 0
    }

    if {$event == "collect fail"} {
        set state get_account
        return 0
    }
    set state end
    return 0
}
```

The **get_pin** procedure defines the parameters of the PIN prompt and collects the PIN. In this procedure:

- The prompt plays an audio file called enter_pin, which is stored in Flash memory.
- The user is allowed to enter information before the prompt message is complete.
- The user is allowed to abort the process by pressing the asterisk (*) key.
- The user must indicate that they have completed their entry by pressing the pound (#) key.
- The input from the user is collected in an array.
- If the collection of information is successful, the script stores the PIN and indicates that the next procedure is to authenticate the information.
- If the user aborts the process, the script indicates that the user should be returned to the account collection procedure.
- If the user does not enter a PIN within the allotted time, the script indicates that this procedure should be started again.
- If the PIN entered by the user is invalid, the script indicates that this procedure should be started again.

```
proc do_get_pin {} {
    global state
    global pin

    set prompt(url) flash:enter_pin.au
    set prompt(interrupt) true
    set prompt(abortKey) *
    set prompt(terminationKey) #
    set patterns(account) .+
    set event [promptAndCollect prompt ReturnInfo patterns ]

    if {$event == "collect success"} {
        set state authenticate
        set pin $info(digits)
        return 0
    }

    if {$event == "collect aborted"} {
        set state get_account
        return 0
    }

    if {$event == "collect fail"} {
        # timeout
```

```
        if {$info(code) == 102} {
            set state get_pin
            return 0
}

        # invalid number
        if {$info(code) == 28} {
            set state get_pin
            return 0
}
    }

    set state end
    return 0
}
```

The **authenticate** procedure passes the information collected to a RADIUS server for authentication and waits for a reply. In this procedure:

- The account number and PIN are passed to the RADIUS server.
- The response from the RADIUS server is interpreted.
- If the authentication is successful, the script indicates that the next state is to continue processing the call.
- If the authentication fails, the script indicates that the next state is to inform the user of the failure.

```
proc do_authenticate {} {
    global state
    global pin
    global account

    set event [authenticate $account $pin info]

    if { $event == "authenticated" } {
         set state authen_pass
         return 0
    }

    if {$event == "authentication failed"} {
        set state authen_fail
        return 0
    }

    set state end
    return 0
}
```

The **get_dest** procedure defines the parameters of the destination prompt and collects the destination phone number. In this procedure:

- The prompt plays an audio file called enter_destination, which is stored in Flash memory.
- The user is allowed to enter information before the prompt message is complete.
- The user is allowed to abort the process by pressing the asterisk (*) key.
- The user must indicate that they have completed their entry by pressing the pound (#) key.
- The destination phone number is collected against the dial plan.
- If the collection of information is successful, the script indicates that the next procedure is to process the call.

- If the user aborts the process, the script indicates that this procedure should be started again.
- If the destination phone number entered by the user is invalid, the script indicates that this procedure should be started again.

```
proc do_get_dest {} {
    global state
    global destination

    set prompt(url) flash:enter_destination.au
    set prompt(interrupt) true
    set prompt(abortKey) *
    set prompt(terminationKey) #
    set prompt(dialPlan) true

    set event [promptAndCollect prompt info ]

    if {$event == "collect success"} {
        set state place_call
        set destination $info(digits)
        return 0
    }

    if {$event == "collect aborted"} {
        set state get_dest
        return 0
    }

    if {$event == "collect fail"} {
        set state get_dest
        return 0
    }
    set state end
    return 0
}
```

The **authen_pass** procedure determines whether the destination phone number has been provided. In this procedure:

- If the incoming dial peer is configured for DID operation and there is an incoming called number ($dnislen is not equal to 0), the destination is set to the destination number and the script indicates that the next state is place_call.
- If the incoming dial peer in not configured for DID operation, the script indicates that the next state is to collect the destination phone number.

```
proc do_authen_pass {} {
    global state
    global destination

    set dnislen [string len [dnis]]

    if { [did] && $dnislen } {
        set destination [dnis]
        set state place_call
    } else {
        set state get_dest
    }
    return 0
}
```

The **place_call** procedure places the call to the specified destination. In this procedure:

- If the call is successful, the script indicates that the next state is to determine whether the call should be timed.
- If the call is unsuccessful, the script indicates that the next state is to process the failed call.

```
proc do_place_call {} {
    global state
    global destination

    set event [placeCall $destination callInfo info]

    if {$event == "active"} {
        set state active
        return 0
    }
    if {$event == "call fail"} {
        set state place_fail
        return 0
    }

    set state end
    return 0
}
```

The **active_notimer** procedure specifies that no time limit is set on the call. The script waits idly for an event.

```
proc do_active_notimer {} {
    global state

    set event [waitEvent]
    while { $event == "digit" } {
        set event [waitEvent]
    }
    set state end
    return 0
}
```

The **active_last_timer** procedure monitors the last few seconds of time allowed for the call to ensure that the user does not exceed their allotted credit time. In this procedure:

- A final timer is set. The duration of the timer is equal to the number of seconds left in the user's account.
- If the call exceeds the timer, the script disconnects the outgoing call leg and indicates that the next procedure is to notify the user that they are out of time.

```
proc do_active_last_timer {} {
    global state

    set event [startTimer [creditTimeLeft] info]
    while { $event == "digit" } {
        set event [startTimer $info(timeLeft) info]
    }
    if { $event == "timeout" } {
        clearOutgoingLeg returnInfo
        set state out_of_time
    } else {
        set state end
    }

    return 0
}
```

The **active_timer** procedure monitors the amount of time used during the call to ensure that the user does not go past their allotted credit time. In this procedure:

- The user's remaining amount of credit is determined.
- If the user has less than ten seconds remaining, the script indicates that the next procedure should be to monitor the last few seconds of the call.
- If the user has more than ten seconds remaining, a timer is set that equals the user's remaining time minus ten seconds.
- If the call exceeds the timer, a warning message (beep.au) is played and the script indicates that the next procedure is to monitor the last few seconds of the call.

```
proc do_active_timer {} {
    global state

    if { [creditTimeLeft] < 10 } {
        do_active_last_timer
        return 0
    }
    set delay [expr [creditTimeLeft] - 10]
    set event [startTimer $delay info]
    while { $event == "digit" } {
        set event [startTimer $info(timeLeft) info]
    }
    if { $event == "timeout" } {
        insertMessage flash:beep.au returnInfo
        do_active_last_timer
    } else {
        set state end
    }

    return 0
}
```

The **active** procedure determines whether there is a time limit on the call. In this procedure:

- If the user has no credit limit, the script indicates that the next procedure is to wait idly for an event.
- If the user has a credit limit, the script indicates that the next procedure is to start an active timer.

```
proc do_active {} {
    global state

    if { ( [creditTimeLeft] == "unlimited") ||
         ([creditTimeLeft] == "uninitialized") } {
     do_active_notimer
    } else {
     do_active_timer
    }
    return 0
}
```

The **out_of_time** procedure plays a message (out_of_time.au) if the user is out of time.

```
proc do_out_of_time {} {
    global state

    set prompt(url) flash:out_of_time.au
    set prompt(playComplete) true
    set event [promptAndCollect prompt info ]
    set state end
    return 0
}
```

The **authen_fail** procedure plays a message (auth_failed.au) if the information that the user provided is not authenticated by the RADIUS server.

```
proc do_authen_fail {} {
    global state

    set prompt(url) flash:auth_failed.au
    set prompt(playComplete) true
    set event [promptAndCollect prompt info ]
    set state end
    return 0
}
```

The **place_fail** procedure plays a tone if the call cannot be placed.

```
proc do_place_fail {} {
    global state

    playFailureTone 5 returnInfo
    set state end
    return 0
}
```

Finally, we put all the procedures together in a main routine. The main routine passes the origination and destination phone numbers (if available) to the RADIUS server for authentication. If the response from the RADIUS server is “authenticated,” the authen_pass procedure is called.

If the response from the RADIUS server is anything other than “authenticated,” the get_account procedure is called.

From there, the main routine reads the state as set by each procedure to determine which procedure to call next.

```
#-------------------------------------------------------
# The main routine begins here.
#

acceptCall

set event [authenticate [ani] [dnis] info]

if {$event != "authenticated"} {
    set state get_account
} else {
    set state authen_pass
}

while {$state != "end"} {
      if {[info proc do_$state] == do_$state}
      do_$state
}
```

Testing and Debugging Your Script

It's important to thoroughly test a script before it is deployed.

To test a script, you must place it on a router and place a call to activate the script. When you test your script, make sure that you test every procedure in the script and all variations within each procedure. Errors that occur in parts of the script that are not tested will not be found. For example, if you have the following:

```
if {[dnis]=="1234"} {
   promptAndcollect prompt info
```

The misspelling of the **promptAndCollect** command will not be detected until the number 1234 is dialed.

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

[no] debug voip ivr [**states** | **error** | **script** | **dynamic**| **all**]

If you specify "error," output is displayed only if an error occurs in the script. If you specify "states," the output provides information about what is happening in an IVR application. If you specify "dynamic" this shows the dynamic prompt play information. If you specify "all," both error and states information is displayed.

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

Note Possible sources of errors are: an unknown or misspelled command (for example, if you misspell promptAndCollect as promptAndcollect), a syntax error (such as, specifying an invalid number of arguments), or executing a command in an invalid state (for example, executing insertMessage when no conference is active). In most cases, an error such as these will cause the underlying infrastructure to disconnect the call and clean up.

Error Handling of TCL IVR Commands

Following are some of the errors that may appear and their causes:

Error Message	Description
`wrong # args: should be "cmd... ..."`	Script exits with error information due to the wrong number of arguments.
`bad args: should be "cmd... ..."`	Script exit with error information due to incorrect argument(s).
`verb-name called in wrong state, ... other detail information...`	Script exit with error information due to verb being called in the wrong state.
`verb-name ignored, ... other detail information`	Command is ignored due to verb being called in the wrong state.

Loading Your Script

To associate an application with your TCL IVR script and load the script, use the following command:

(config)#call application voice *script_name url* [*parameter value*]

In this command:

- *script_name* specifies the name of the TCL application that the system is to use for the calls configured on the inbound dial peer. Enter the name to be associated with the TCL IVR script.
- *url* is the pathname where the script is stored. Enter the pathname of the storage location first and then the script filename. TCL IVR scripts can be stored in Flash memory or on a server that is acceptable using a URL, such as a TFTP server.
- *parameter value* allows you to configure values for specific parameters, such as language or PIN length. For more information about possible parameters, see the "Debit Card for Packet Telephony on Cisco Access Platforms" document on CCO.

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

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

If you modify your script, you can reload it using only the script name as shown below:

(config)#call application voice load *script_name*

Associating Your Script with an Inbound Dial Peer

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

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

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

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

In these commands:

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

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

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

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

Displaying Information About IVR Scripts

To view a list of the voice applications that are configured on the router, use the **show call application voice** command. A one-line summary of each application is displayed. The description includes the names of the audio files the script plays, the operation of the interrupt keys, what prompts are used, and the caller interaction.

#show call application voice [[*name*] | [**summary**]]

In this command:

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

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

```
router#show call app voice clid_authen_collect
Idle call list has 0 calls on it.
Application clid_authen_collect
    The script is compiled into the image
    It has 0 calls active.

The TCL Script is:
------------------
# clid_authen_collect.tcl
#---------------------------------
# September 1998, David Ramsthaler
#
# Copyright (c) 1998, 1999 by cisco Systems, Inc.
# All rights reserved.
#---------------------------------
# Mimic the clid_authen_collect script in the SP1.0 release.
#
# It authenticates using (ani, dnis) for (account, password). If
# that fails, it collects account and pin number, then authenticates
# using (account, pin).
#
# If authentication passes, it collects the destination number and
# places the call.
#
# The main routine is at the bottom. Start reading the script there.
#
proc do_get_account {} {
    global state
    global account

    set prompt(url) flash:enter_account.au
    set prompt(interrupt) true
    set prompt(abortKey) *
    set prompt(terminationKey) #
    set patterns(account) .+
    set event [promptAndCollect prompt info patterns ]

    if {$event == "collect success"} {
        set state get_pin
        set account $info(digits)
        return 0
    }

    if {$event == "collect aborted"} {
        set state get_account
```

```
        return 0
    }

    if {$event == "collect fail"} {
        set state get_account
        return 0
    }
    set state end
    return 0
}

proc do_get_pin {} {
    global state
    global pin

    set prompt(url) flash:enter_pin.au
    set prompt(interrupt) true
    set prompt(abortKey) *
    set prompt(terminationKey) #
    set patterns(account) .+
    set event [promptAndCollect prompt info patterns ]

    if {$event == "collect success"} {
        set state authenticate
        set pin $info(digits)
        return 0
    }

    if {$event == "collect aborted"} {
        set state get_account
        return 0
    }

    if {$event == "collect fail"} {
        # timeout
        if {$info(code) == 102} {
            set state get_pin
            return 0
        }

        # invalid number
        if {$info(code) == 28} {
            set state get_pin
            return 0
        }
    }

    set state end
    return 0
}

proc do_authenticate {} {
    global state
    global pin
    global account

    set event [authenticate $account $pin info]

    if { $event == "authenticated" } {
        set state authen_pass
        return 0
    }

    if {$event == "authentication failed"} {
        set state authen_fail
```

```
        return 0
    }

    set state end
    return 0
}

proc do_get_dest {} {
    global state
    global destination

    set prompt(url) flash:enter_destination.au
    set prompt(interrupt) true
    set prompt(abortKey) *
    set prompt(terminationKey) #
    set prompt(dialPlan) true

    set event [promptAndCollect prompt info ]

    if {$event == "collect success"} {
        set state place_call
        set destination $info(digits)
        return 0
    }

    if {$event == "collect aborted"} {
        set state get_dest
        return 0
    }

    if {$event == "collect fail"} {
        set state get_dest
        return 0
    }
    set state end
    return 0
}

proc do_authen_pass {} {
    global state
    global destination

    set dnislen [string len [dnis]]

    if { [did] && $dnislen } {
        set destination [dnis]
        set state place_call
    } else {
        set state get_dest
    }
    return 0
}

proc do_place_call {} {
    global state
    global destination

    set event [placeCall $destination callInfo info]

    if {$event == "active"} {
        set state active
        return 0
    }
    if {$event == "call fail"} {
        set state place_fail
```

```
        return 0
    }

    set state end
    return 0
}

proc do_active_notimer {} {
    global state

    set event [waitEvent]
    while { $event == "digit" } {
        set event [waitEvent]
    }
    set state end
    return 0
}

proc do_active_last_timer {} {
    global state

    set event [startTimer [creditTimeLeft] info]
    while { $event == "digit" } {
        set event [startTimer $info(timeLeft) info]
    }
    if { $event == "timeout" } {
        clearOutgoingLeg retInfo
        set state out_of_time
    } else {
        set state end
    }

    return 0
}

proc do_active_timer {} {
    global state

    if { [creditTimeLeft] < 10 } {
        do_active_last_timer
        return 0
    }
    set delay [expr [creditTimeLeft] - 10]
    set event [startTimer $delay info]
    while { $event == "digit" } {
        set event [startTimer $info(timeLeft) info]
    }
    if { $event == "timeout" } {
        insertMessage flash:beep.au retInfo
        do_active_last_timer
    } else {
        set state end
    }

    return 0
}

proc do_active {} {
    global state

    if { ( [creditTimeLeft] == "unlimited") ||
         ([creditTimeLeft] == "uninitialized") } {
            do_active_notimer
    } else {
            do_active_timer
```

```
    }
    return 0
}

proc do_out_of_time {} {
    global state

    set prompt(url) flash:out_of_time.au
    set prompt(playComplete) true
    set event [promptAndCollect prompt info ]
    set state end
    return 0
}

proc do_authen_fail {} {
    global state

    set prompt(url) flash:auth_failed.au
    set prompt(playComplete) true
    set event [promptAndCollect prompt info ]
    set state end
    return 0
}

proc do_place_fail {} {
    global state

    playFailureTone 5 retInfo
    set state end
    return 0
}

#---------------------------------------
# And here is the main loop
#

acceptCall

set event [authenticate [ani] [dnis] info]

if {$event != "authenticated"} {
    set state get_account
} else {
    set state authen_pass
}

while {$state != "end"} {
    puts "cid([callID]) running state $state"
    if {$state == "get_account"} {
        do_get_account
    } elseif {$state == "get_pin"} {
        do_get_pin
    } elseif {$state == "authenticate"} {
        do_authenticate
    } elseif {$state == "get_dest"} {
        do_get_dest
    } elseif {$state == "place_call"} {
        do_place_call
    } elseif {$state == "active"} {
        do_active
    } elseif {$state == "authen_fail" } {
        do_authen_fail
    } elseif {$state == "authen_pass" } {
        do_authen_pass
    } elseif {$state == "place_fail"} {
```

```
            do_place_fail
        } elseif {$state == "out_of_time"} {
            do_out_of_time
        } else {
            break
        }
    }
```

Using URLs in IVR Scripts

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

Note Flash memory is limited to 32 entries, so it may not be possible to copy all your audio files into Flash memory.

URLs for Loading the IVR Script

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

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

URLs for Loading Audio Files

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

- For static prompts (used with the **playPrompt**, **insertMessage**, and **promptAndCollect** commands), you can use the IFS-supported URLs as described in the “URLs for Loading the IVR Script” on page 20.
- For dynamic prompts (used in the **playPrompt** command), the URL is created by the software using information from the **setVariable language** and **setLocation** commands and the language CLI configuration command.

Tips for Using Your TCL IVR Script

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

1 How do I get information from my RADIUS server to the TCL IVR script?

After you perform an authentication and authorization, you can use the **getVariable** command to obtain the credit amount, credit time, and cause codes maintained by the RADIUS server. See the "getVariable" on page 38.

2 How do I keep the network from charging for the incoming call if there is no answer?

Although gateways do not control the charges for each call, in most networks you are only charged for completed calls. Therefore, if the incoming network does not detect a completed call, you are not charged. The **placeCall** command accepts an incoming call only if it succeeds in placing the call. At that point the incoming network will detect the completed call and charges will begin to accrue.

Normally, if the script is going to play prompts and collect digits, it will need to accept the incoming call by running the **acceptCall** command. If the script does not accept the incoming call, there may be a relatively short timer on the incoming network.

3 What happens if my script encounters an error?

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

If the script exits with an error and IVR debugging is on (as described in "Testing and Debugging Your Script" on page 13), the location of the error in the script is displayed at the command line.

TCL IVR Command Reference

This section provides an alphabetical listing of the new TCL IVR commands. 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
- Whether the command is blocking (meaning the command has some external interaction) or nonblocking
- Example of how the command can be used

For information about how returns and events are defined in the code, see "Value Definitions" on page 61.

Notes For the commands that return information, the returnInfo array often contains useful information. However, if the command is returned because the destination (called) party hangs up, then the returnInfo(**code**) is the failure code.

If you specify an invalid number of arguments for a command, an error will occur and the script will fail.

---

acceptCall

The **acceptCall** command is used at the beginning of the main routine in a TCL script. It sends setup acknowledgement, call proceeding, and call connect messages to the incoming call leg. The Gateway is responsible for translating these messages into the appropriate protocol messages (depending on the call leg) and sending them to the caller.

Syntax

acceptCall

Arguments

None

Return Values

None

Blocking or Non Blocking

Non blocking

Example

```
# Accept a call and then prompt redialer for a pattern
acceptCall
set prompt(url) flash:redialer_tone.au
set prompt(interrupt) true
set patterns(accountAndPW) \\*\\*.+#\\*\\*.+#
set event [promptAndCollect prompt returnInfo patterns]
```

Usage Notes

- On an ISDN line, this command sends a setup acknowledgment. On an analog line, this command has no external effect. However, it does stop the internal timers on that call leg.
- If the call has already been connected, this command is ignored. If IVR state debugging is on (see the “Testing and Debugging Your Script” on page 13), a message is displayed that indicates that the command was ignored.
- The command is ignored if the incoming call has already been connected.
- If there is no incoming call, the script will exit with error information.
- The system tracks whether a setup acknowledgement or call proceeding has already been sent, and will not send another.

ani

The **ani** command returns the call origination number, which it obtains from the incoming callInfo data structure.

Syntax

ani

Arguments

None

Return Values

This command returns the origination number (as a decimal string) or a NULL string if the number is not available. The origination number is the number provided by the incoming call leg.

Blocking or Non Blocking

Non blocking

Example

```
# Authenticate using origination number as the user account number
set event [authenticate [ani] [dnis] info]
```

Usage Notes

- On some incoming calls, such as those coming over ISDN, the calling number might not be provided. If it is not and the you have configured the router such that calls from this port have a configured calling number (using the **answer-address** command), the system can attempt to determine the calling number based on the information in the inbound dial-peer and the incoming port number. For more information on the answer-address command, see the *Cisco IOS Voice, Video, and Home Applications Command Reference*.
- If there is no incoming call, the script will exit with error information.

authenticate

The **authenticate** command validates the authenticity of the user. This command sends the account number and password to the appropriate server for authentication and waits for a response. In many cases, ani is used as the account number and dnis (or a NULL value) is used as the password.

Syntax

authenticate *account password* [**av-send**] *returnInfo*

Arguments

The arguments of this command are:

- *account*—The user's account number for the AAA request.
- *password*—The user's password (or PIN) for the AAA request.
- **av-send**—If it is present, it is an associate array containing standard AV or VSA pairs to be sent along.
- *returnInfo*—The name of an associative array where information about the result is stored.

Return Values

This command returns the following:

- The event returned is:
  - authenticated—The authentication was successful.
  - authentication failed—The authentication failed.
- If the authentication fails, the AAA failure code is stored in returnInfo(**code**).
- If the caller hangs up or the call becomes disconnected, the authentication will not be valid.

Blocking or Non Blocking

Blocking

Example

```
# Authenticate using origination number as the account number
set event [authenticate [ani] [dnis] info]
if {$event == "authentication failed"} {
   puts authentication failed with code $info(code)
   exit
}
```

Usage Notes

Caution The authorize command will return erroneous VSA values with Cisco IOS 12.1T and higher versions such as 12.2M, 12.2T, 12.3M, and 12.3T.

- Typically a RADIUS server is used for authentication, but any AAA-supported method can be used.

- If IVR state debugging is on (see the “Testing and Debugging Your Script” on page 13), the account number and password are displayed.
- Account numbers and PINs are truncated at 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 the *Named Method Lists for AAA Authorization and Accounting* document located at http://www.cisco.com/univercd/cc/td/doc/product/software/ios113ed/113t/113t_3/aaalists.htm
- To define av-send, use the command **set av-send attrName [index] value**.

Note Cisco IOS Release 12.1(2) is the first release incorporating the argument **av-send**.

— attrName—Currently, only two IVR-specific attributes are supported: “h323-ivr-out” and “h323-credit-amount”. Refer to the table in the **getVariable** command description for more information on these types.

— index—An optional integer index starting from 0, used to distinguish multiple values for a single attribute.

Example 1:

```
set av-send(h323-credit-amount) 25.0
```

Example 2: (Using two values for h323-ivr-out)

```
set av-send(h323-ivr-out,0) "payphone:true"

set av-send(h323-ivr-out,1) "creditTime:3400"
```

authorize

The **authorize** command sends an AAA authorization request and waits for a response. Typically, this command is used to provide additional information to the RADIUS server (destination and origination numbers) after a user has been successfully authenticated. It sends the authorization request to the RADIUS server after the IVR application prompts the user to enter the destination number.

Syntax

authorize *account password destination ani* [**av-send**] *returnInfo*

Arguments

The arguments of this command are:

- *account*—The user's account number.
- *password*—The user's password (or PIN).
- *destination*—The call destination number.
- *ani*—The origination number.
- **av-send**—Indicates that the values that were specified with the **set av-send** command should be included in the authorization.
- *returnInfo*—The name of an associative array where information about the result is stored.

The account, password, and returnInfo arguments are the same as those specified in the **authentication** command. The destination is the destination, or called, number and the ani is the origination, or calling, number. The destination and ani fields are used to provide additional information to the external server.

Return Values

This command returns the following:

- The event returned is:
  - authorized—The authorization was successful.
  - authorization failed—The authorization failed.
- If the authentication fails, the AAA failure code is stored in returnInfo(**code**).
- The set of VSAs from the AAA return will be available through the **getVariable** command.
- If the caller hangs up or the call becomes disconnected, the authorization is not valid.

Blocking or Non Blocking

Blocking

Example

```
# authorize the call, and get the available credit time.
set event [authorize $account $pin $destination [ani] info]

if {$event == “authorized”} {
   set num [getVariable aaa h323-credit-amount CreditAmount]
}
```

Usage Notes

Caution The authorize command will return erroneous VSA values with Cisco IOS 12.1T and higher versions such as 12.2M, 12.2T, 12.3M, and 12.3T.

- If IVR state debugging is on (see the “Testing and Debugging Your Script” on page 13), the account number, password, and destination are displayed.
- Account numbers, PINs, and destination numbers are truncated at 32 characters, the E.164 maximum length.
- In addition to the failure code, which is stored in the returnInfo array as returnInfo(**code**), return codes are generated by the external RADIUS server after an authorization request has been processed. These return codes contain information about the account and the destination called, and indicate what actions the application can carry out in the event of a failed authorization. You can retrieve these return codes using **getVariable aaa** h323-return-code returnInfo.
- You can use the **aaa authentication login** and **radius-server** commands to configure a number of RADIUS parameters. For more information, see the *Named Method Lists for AAA Authorization and Accounting* document located at http://www.cisco.com/univercd/cc/td/doc/product/software/ios113ed/113t/113t_3/aaalists.htm
- If there is no incoming call, the script exits with error information.
- To define av-send, use the command **set av-send (attrName [,index]) value**.

Note Cisco IOS Release 12.1(2) is the first release incorporating the argument **av-send**.

  - attrName—The standard attribute name or VSA attribute name string defined by AAA of Cisco IOS. Currently, only two IVR-specific VSAs are supported: “h323-ivr-out” and “h323-credit-amount”.
  - index—An optional integer index starting from 0, used to distinguish multiple values for a single attribute.

  Example 1:

```
set av-send(h323-credit-amount) 25.0
```

  Example 2: (Using two values for h323-ivr-out)

```
set av-send(h323-ivr-out,0) "payphone:true"

set av-send(h323-ivr-out,1) "creditTime:3400"
```

callID

The **callID** command is used for debugging purposes only. It can be used in problem determination for a particular call. This command returns the call ID (which is used in the IOS debug messages) of the incoming or outgoing call leg. This call ID has no meaning outside the router that is handling the call.

Syntax

callID [**outgoing**]

Arguments

The argument of this command is:

- **outgoing**—(Optional) Indicates that the command should return the callID of the outgoing call leg.

Return Values

This command returns an integer greater than zero, which represents the call ID of the incoming or outgoing call leg. If you specify outgoing and there is no outgoing call leg, zero is returned.

Blocking or Non Blocking

Non blocking

Example

```
# Display the call IDs in a debug message
puts “Incoming callID [callID] connected to outing callID [callID outgoing]”
```

Usage Notes

Each call leg is assigned a call ID. The Cisco IOS software begins assigning call IDs at 1 when the system is rebooted. The call ID is increment for each call leg and will eventually wrap.

callProceeding

The **callProceeding** 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.

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

Syntax

callProceeding

Arguments

None

Return Values

None

Blocking or Non Blocking

Non blocking

Example

```
# Send call proceeding while placing the call
callProceeding

# we have not accepted the call yet. placeCall will place the outbound
# call, conference the two call legs, and accept the inbound call leg.
#
set callInfo(destinationNum) $destination
set event [placeCall $destination callInfo info]
```

Usage Notes

- If a call proceeding message has already been sent (for example, if a callProceeding or acceptCall command has already been executed), this command is ignored. If IVR debugging is on (see the “Testing and Debugging Your Script” on page 13), the fact that the command has been ignored will be displayed.
- If there is no incoming call, the script will exit with error information.

clearOutgoingLeg

The **clearOutgoingLeg** command deletes the conference and disconnects the outgoing call leg.

Syntax

clearOutgoingLeg *returnInfo*

Arguments

The argument of this command is:

- *returnInfo*—The name of an associative array where information about the result is stored.

Return Values

This command returns a “disconnect done outgoing” event, even if no outgoing call leg exists.

Blocking or Non Blocking

Blocking

Example

```
# If time runs out, clear the outgoing call and play out of time message
if {$event == “timeout”} {
   clearOutgoingLeg returnInfo
   set prompt(url) flash:out_of_time.au
   set event [promptAndCollect prompt info]
}
```

Usage Notes

The command is ignored if the outgoing call is not connected.

conferenceCreate

The **conferenceCreate** command creates a conference between the call legs. This command ignores any digits entered by the caller.

Syntax

conferenceCreate *returnInfo*

Arguments

The argument of this command is:

- *returnInfo*—The name of an associative array where information about the result is stored.

Return Values

This command returns a “conference created” event. If the called party hangs up during the conferenceCreate, an “outgoing disconnected” event is returned and the returnInfo(**code**) will indicate the cause.

Blocking or Non Blocking

Blocking

Example

```
# Delete the conference, play a message, then recreate it
set event [conferenceDelete returnInfo]
if {$event!= “conference deleted”} {
   exit 0
}
set prompt(url) message.au
promptAndCollect prompt info
set event [conferenceCreate returnInfo]
if {$event!= “conference created”} {
   exit
}
```

Usage Notes

If this command is invoked while a conference is already established, the command is ignored. If two call legs do not exist, the script returns an error code, which causes the underlying infrastructure to disconnect the call and clean up.

conferenceDelete

The **conferenceDelete** command deletes a conference between call legs. It ignores digits and will abort and return if it receives any event other than the expected one.

Syntax

conferenceDelete *returnInfo*

Arguments

The argument of this command is:

- *returnInfo*—The name of an associative array where information about the result is stored.

Return Values

This command returns a “conference deleted” event, even if a conference does not exist. If the outgoing call leg disconnected, “outgoing disconnected” is returned.

Blocking or Non Blocking

Blocking

Example

```
# Delete the conference, play a message, then recreate it
set event [conferenceDelete returnInfo]
if {$event != “conference deleted”} {
   exit 0
}
set prompt(url) message.au
promptAndCollect prompt info
set event [conferenceCreate returnInfo]
if { $event != “conference created” } {
   exit 0
}
```

Usage Notes

If the destination party disconnects, the cause code is saved internally but is not returned with this command.

creditTimeLeft

The **creditTimeLeft** command returns the amount of credit time available for this call.

Syntax

creditTimeLeft [**-settlement** | **-aaa**]

Arguments

The arguments of this command are:

- **-settlement**—(Optional) Requests the Open Settlement Protocol (OSP) credit time available.
- **-aaa**—(Optional) Requests the authentication, authorization, and accounting (AAA) credit time available.

If no argument is specified, the lesser of the two credit times is requested.

Return Values

This command returns one of the following:

- An integer greater than or equal to 0, indicating the number of seconds of credit available. If 0 is returned, there is no credit time available.
- “uninitialized” or “unlimited,” indicating that there is no credit limit. A return value of “uninitialized” means that no credit time was returned from the authorize command and OSP settlement was not used to get a credit time. If “unlimited” is returned, no credit line has been set.

If no argument is specified, the lesser of the OSP and AAA values is returned. If the two values are an integer and “uninitialized”, the integer is returned.

Blocking or Non Blocking

Non blocking

Example

```
# wait for the credit time we have
set event [startTimer [creditTimeLeft] returnInfo]
while { $event == “digit” } {
   set event [startTimer [creditTimeLeft] returnInfo]
}
```

Usage Notes

- If you are using the **aaa** option, the command **authorize** must be run before using **creditTimeLeft**, so that the **creditTimeLeft** values will be initialized.
- If no argument is specified and the values returned by AAA or OSP are “unlimited” and “uninitialized”, the value returned by this command will be “unlimited.”
- The following values are defined for **creditTimeLeft** returns:
  - CT_INVALID—“uninitialized”
  - CT_UNLIMITED—“unlimited”
- If both the AAA and OSP servers are down, “uninitialized” is returned.

did

The **did** command determines whether the incoming dial peer is configured for DID operation.

Syntax

did

Arguments

None

Return Values

This command returns either a 0 (the dial peer is not configured for DID) or a 1 (the dial peer is configured for DID).

Blocking or Non Blocking

Non blocking

Example

```
# If DID, just place the call
if {[did]} {
    set destination [dnis]
} else {
#   play a dial tone, and collect against the pattern
    playTone Dial
    set prompt(dialPlan) true
    set event [promptAndCollect prompt returnInfo]
    if {event != “collect success”} {
       puts “Call [callID] got event $event collecting destination”
    exit 0
    }
    set destination returnInfo(digits)
}
```

Usage Notes

If there is no incoming call, the script will exit with error information.

dnis

The **dnis** command returns the destination phone number.

Syntax

dnis

Arguments

None

Return Values

This command returns the destination number, if it was provided with the incoming call. If no destination number was provided with the incoming call, this command returns a null value.

Blocking or Non Blocking

Non blocking

Example

```
# Authenticate using destination number as the user password
set event [authenticate [ani] [dnis] info]
```

Usage Notes

- With the **dnis** command, number expansion does not operate on the number before it is given to the script. Number expansion occurs during the **placecall** command.
- If there is no incoming call, the script will exit with error information.

exit

The **exit** command exits the script, and then clears the call and cleans up the data structures. You can use this command to set the cause code for a call.

Syntax

exit [*code*]

Arguments

The argument of this command is:

- *code*—(Optional) Causes a code to be displayed in a debug output message if the IVR state debugging is on (see the "Testing and Debugging Your Script" on page 13). If no code is provided, zero will be shown in the debug output. If the value of code is a positive integer, the cause code of the call leg is set to that value.

  The cause code is a Q.931 cause code, which is used in the disconnect message and stored in the Call Data Record (CDR) for the call.

Return Values

None

Blocking or Non Blocking

Non blocking

Example

```
# set event [placeCall destination callInfo info]

if {$event != active} {
    puts "Call [callID] got $event when placing call"
    exit 0
}
```

Usage Notes

If the inbound call leg is not yet connected, setting a cause code can change what the caller hears. It may be a busy signal, a fast busy signal, or silence.

failureCode

The **failureCode** command returns the last failure event and code.

Syntax

failureCode *returnInfo*

Arguments

The argument of this command is:

- *returnInfo*—The name of an associative array where information about the result is stored.

Return Values

The return value is the return event if the last failure event was “authentication failed”, “collect fail”, or “call fail”. In these cases, returnInfo(**code**) is failure code from that event. Otherwise (if there is no previous failure event), the return value is “success”, and returnInfo(**code**) is “0”.

Blocking or Non Blocking

Non blocking

Example

```
# get the last failure event and code
set event [failureCode returnInfo]
if {$returnInfo(code) == 17} {
   set destination $forwardOnBusy
}
```

getVariable

The **getVariable** command returns the value of a variable based on its type and name.

Syntax

getVariable {**callInfo** | **config** | **local** | **radius** | **aaa**} *name* [*returnInfo*]

Arguments

The arguments of this command are:

- **callInfo**—Obtained from the call information.
- **config**—Obtained from the CLI configuration.
- **local**—Obtained from internal variables.
- **radius**—Obtained from the RADIUS server. The argument **radius**, though currently supported, has been obsoleted by **aaa**. It is strongly recommended that you use **aaa** and not **radius**.
- **aaa**—Obtained from the AAA server. If **aaa** is specified you should use one of the attribute names listed the table below.
  - *returnInfo*—(Option only when **aaa** is chosen) Name of an associative array to return information when the type is **aaa**.
- *name*—Name of the variable to retrieve. The possible name depends on the specified type, see the following table of possible command options for **getVariable**.

Variable Type	Variable Name	Variable Description
callInfo	ani	The origination (calling) number is returned.
	callIDin	The incoming call ID is returned. “0” is the return value if there is no incoming call ID.
	callIDout	The outgoing call ID is returned. “0” is the return value if no outgoing call ID.
	callIPin	The incoming IP leg's address is returned. Null is returned if the incoming leg is not an IP leg.
	did	The value of the DID flag (0 or 1) is returned.
	dnis	The destination (called) number is returned.
config	uidLen	The account length is returned.
	pinLen	The PIN length is returned.
	warningTime	The time to play the warning message is returned.
	retryCount	The number of retries for entering digits is returned.
	redirectNumber	The number for redirection of a call is returned. This variable can return a value of “not configured”.
local	creditTimeLeft	The length of credit time remaining is returned.
	language	The language selected is returned.
radius[1]	creditAmount	The credit amount remaining in the account is returned.
	creditTime	The credit time remaining in the account is returned.
	promptID	The ID of the prompt is returned.
	redirectNumber	The number for redirection of a call is returned.
	redirectIP	The IP address for the preferred route is returned.
	preferredLang	The language that the billing system returns as the preferred language of the end user. Currently, three languages are supported; en (english), sp (spanish) and ch (mandarin). You can define additional languages as needed.
	timeAtDest	The current time at the destination.
	returnCode	This information is returned only after an authorization command is issued. It returns either a numerical value or “Unknown variable name.” The numerical value indicates what action the IVR application should take, namely to play a particular audio file to inform the end user of the reason for the failed authorization. If “Unknown variable name” is returned, this indicates that the external RADIUS server is out of service.

Variable Type	Variable Name	Variable Description
aaa	h323-ivr-in	A generic VSA for the billing server to send any information to the gatway in the form of an avpair like “color:blue” or “advprompt:rtsp://www.cisco.com/rtsp/areyouready.au
	h323-ivr-out	A generic VSA for the gateway to send any information to the billing server in the form of an avpair like “color:blue” or “advprompt:rtsp://www.cisco.com/rtsp/areyouready.au
	h323-credit-amount	The credit amount remaining in the account is returned.
	h323-credit-time	The credit time remaining in the account is returned.
	h323-prompt-id	The ID of the prompt is returned.
	h323-redirect-number	The number for redirection of a call is returned.
	h323-redirect-ip-addr	The IP address for the preferred route is returned.
	h323-preferred-lang	The language that the billing system returns as the preferred language of the end user. Currently, three languages are supported; en (english), sp (spanish) and ch (mandarin). You can define additional languages as needed.
	h323-time-and-day	The current time and day at the destination.
	h323-return-code	This information is returned only after an authorization command is issued. It returns either a numerical value or “Unknown variable name.” The numerical value indicates what action the IVR application should take, namely to play a particular audio file to inform the end user of the reason for the failed authorization. If “Unknown variable name” is returned, this indicates that the external AAA-server is out of service.
	h323-billing-model	Indicates the billing model used for the call. Initial values: 0=Credit, 1=Debit. Note: The debit card application will assume a Debit billing model.

1 The argument **radius**, though currently supported, has been obsoleted by **aaa**. It is strongly recommended that you use **aaa** and not **radius**.

Note If the **aaa** variable returns “0”, this indicates that there is no VSA match to the name returned.

Return Values

This command returns the value of the variable, as indicated in the preceding table. If you specify a variable name other than one listed in the preceding table, the command returns “Unknown variable name.” Following is an example of how AAA information is returned for the attribute h323-ivr-in:

```
returnInfo(0) payphone:true
returnInfo(1) color:blue
```

Blocking or Non Blocking

Non blocking

Example Script

```
set event [authorize $account $pin $destination $ani info]
set num [getVariable aaa h323-ivr-in ivrIn]
if {$num != 0} {
## We got values back for h323-ivr-in, $num=2
## ivrIn(0)="payphone:true"
## ivrIn(1)="whateverValue"
## Do something
}

set num [getVariable aaa h323-credit-amount creditAmount]
if {$num != 0} {
## We got creditAmount back, $num=1
## creditAmount(0)=100.00
## Do something
}

set num [getVariable aaa xxx returnInfo]
## $num=0, because there is no VSA returned with a name matched "xxx"
```

Usage Notes

- The script will exit with error information if a wrong variable type is used.
- If a wrong variable name is used with the variable types callInfo, config, or local, the script will exit with error information.
- If a wrong variable name is used with the variable types aaa or radius, the verb will return Uknown variable name.
- If you want to obtain additional AAA information, do the following:
  - On the AAA server, populate a VSA with following string:

    `h323-ivr-in=`*`value`*

    where *value* is the value of the variable that you want the AAA server to provide to the script. For example: `h323-ivr-in=payphone:true`
  - In the script, you can obtain the information by using the following statement:

    ```
    getVariable aaa h323-ivr-in returnInfo
    ```

For more information about using VSAs to transmit information between the Voice Gateway and the AAA server, see the *Vendor-Specific Attributes Implementation Guide*.

Note The getVariable command does not access the AAA server. It allows the script to access information returned in the last authorization or authentication interaction with the AAA server.

---

insertMessage

The **insertMessage** command inserts a message to the originating (calling) party. This command blocks until the message completes. Digits are ignored. The destination (called) party will not hear anything (the tone or the calling party) while the message is playing.

Syntax

insertMessage *url returnInfo*

Arguments

The arguments of this command are:

- *url*—The name and location of an .au file (in G.711 8-bit ulaw encoding).
- *returnInfo*—The name of an associative array where information about the result is stored.

Return Values

This command returns “play complete.” However, “outgoing disconnected” is returned if the called party hangs up.

This command returns “play fail” if the message fails to play.

Blocking or Non Blocking

Blocking

Example

```
# Give the caller a warning 10 seconds before cutting him off
set event [startTimer [expr creditTime-10] retinfo]
if {$event == “Timeout”}
   insertMessage tenSecondsLeft.au returnInfo
   if { $event != “play complete} exit 0
   startTimer 10
   exit 0
}
```

Usage Notes

If no conference is active when this command is invoked, the script returns an error code, which causes the underlying infrastructure to disconnect the call and clean up.

insertTone

The **insertTone** command inserts a beep to the originating (calling) party. The tone is actually a 60 msec ring. The destination (called) party will not hear the tone. This command is typically used to warn the caller 10 seconds before the call is cut off.

Syntax

insertTone

Arguments

None

Return Values

None

Blocking or Non Blocking

Non blocking

Example

```
# If a timeout occurs, tell the caller with a beep
if { $event == "timeout" } {
   insertTone
}
```

Usage Notes

- If a tone was already playing due to a **playTone** command, that tone will be halted.
- If no conference exists, the script will exit with error information.

For information about how tones are defined in the code, see the “Tones” on page 62.

placeCall

The **placeCall** command places a call to the specified destination.

Syntax

placeCall *destination info returnInfo*

Arguments

The arguments of this command are:

- *destination*—The number to look up in the dialing plan to place the call.
- *info*—An array of arguments describing how to place the call.
  - *info*(**originationNum**)—The origination number to use.
  - *info*(**accountNum**)—The account number to use.
  - *info*(**destinationNum**)—The called number to use.
  - *info*(**redirectNum**)—The redirection number to use.
  - *info*(**pinNum**)—The PIN to use.
- *returnInfo*—The name of an associative array where information about the result is stored.

Return Values

This command returns “active” or “call fail.” If the call fails, the failure code is stored in returnInfo(**code**).

Blocking or Non Blocking

Blocking

Example

```
# Place a call, setting an account number, and check the return
set callInfo(accountNum) $account
set ev [placeCall $destination callInfo info]
if {$ev == “active”} {
   set state active
}
if {$ev == "call fail"} {
   set state place_fail
}
```

Usage Notes

- Even if **placecall** returns “active”, the outgoing call leg is not created if the dial peer that matches the call has a loopback session target configured; the outgoing callID cannot be retrieved.
- If there is no incoming call, the script will exit with error information.
- If there is already an active outgoing call, the script will exit with error information.

- You can use the **placecall** command to set the OSP credit time. If the script places a call to a number that uses a dial-peer which is configured for OSP, **placecall** automatically initiates the OSP transaction, which returns a credit time.
- Information configured in the info array overrides information received from the incoming call. If an info array value is not specified, information from the incoming callInfo is used.
- If you do not configure the **destinationNum**, the called number in the outbound call is generated using the standard IOS algorithm (which uses the destination and the matched digits in the dial-peer). Also, you can set destination to one number and still set a different number (with the **destinationNum** field) to be used in the called number field.

The following arguments are defined in the code for the **placeCall** command:

- PLACE_DEST—“destinationNum”
- PLACE_CALLING—“originationNum”
- PLACE_ACCOUNT—“accountNum”
- PLACE_PIN—“pinNum”
- PLACE_REDIRECT—“redirectNum”

playFailureTone

The **playFailureTone** command plays a tone according to the last failure event and code. This command blocks until either the number of seconds elapses or an event occurs. The tone played is busy if the last failure code is Normal, unspecified, or busy. Otherwise, the tone played is congestion. This command is typically used to give the caller the appropriate tone (fast busy, normal busy, and so forth) on failure.

Syntax

playFailureTone *numSeconds returnInfo*

Arguments

The arguments of this command are:

- *numSeconds*—The number of seconds the tone should be played. Possible values are 1 to 4 million seconds.
- *returnInfo*—The name of an associative array where information about the result is stored.

Return Values

This command returns “timeout” or an event. The value is stored in returnInfo(timeLeft). If a timeout occurs, returnInfo(timeLeft) is 0.

Blocking or Non Blocking

Blocking

Example

```
# play failure tone for 5 seconds
playFailureTone 5 returnInfo
```

Usage Notes

- This command does not turn off the tone, even after the timer expires. However, scripts typically play the failure tone and then disconnect, so the tone stops when the call is cleared.
- If there is no incoming call, the script will exit with error information.

For information about how tones are defined in the code, see the “Tones” on page 62.

playPrompt

The **playPrompt** command plays a prompt to the incoming call leg. This command is similar to the **promptAndCollect** command, except this command allows dynamic prompting.

Syntax

playPrompt *prompt_array_name returnInfo* [*promptInfo*]

Arguments

The arguments of this command are:

- *prompt_array_name*—An associative array of arguments describing how to play the prompt.
  - *prompt_array_name*(**insertMessage**)—Indicates to insert message in a conference if set to True.
  - *prompt_array_name*(**interrupt**)—Allows the user to interrupt the prompt and enter digits before the prompt completes playing.
  - *prompt_array_name*(**abortKey**)—Specifies a key to allow the user to abort the prompt.
  - *prompt_array_name*(**terminationKey**)—Specifies a key to terminate pattern collection.
  - *prompt_array_name*(**dialPlan**)—Specifies to collect digits against the dial plan.
  - *prompt_array_name*(**dialPlanTerm**)—Specifies to collect digits against the dial plan but not to return until a timeout occurs while collecting digits.
  - *prompt_array_name*(**playComplete**)—Specifies to return on play completion (not to wait for digit collection). The default value is True, but if any of the arguments **dialPlan**, **dialPlanTerm**, or **maxDigits** are set, these may override it, changing the **playComplete** value to False.
  - *prompt_array_name*(**maxDigits**)—Specifies the maximum number of digits to be collected.
- *returnInfo*—The name of an associative array where information about the result is stored.
- *promptInfo*—(Optional) Information about the prompt to play. For static prompts this can be a pointer to the location of the audio files.
  - URL—The location of an audio file. The URL must contain a colon or the code treats it as a file name, adding .au to the location.
    For example: tftp://223.255.254.254/lang_en/en_welcome.au
  - *name*.au—The name of an audio file. The audio file location values will be appended to the *name*.au. The file name cannot contain a colon, or it will be treated as a URL.

  For dynamic prompts, the following fields are used:

  - %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).
  - %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 msecs).

- %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 msec.
- %n*num*—Plays a string of digits without pauses.
- %i*id*—Plays an announcement. The *id* must be 2-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.

Return Values

- “play complete”—The message is played, but no digit collection is needed (when playComplete is set).
- “play fail”—The message fails to play.
- “collect success”—The message is played and the digit collection succeeds.
- “collect fail”—The message is played, but the digit collection does not succeed.
- “collect aborted”—The message is played, but the digit collection does not succeed.
- “no conference exits”—The insertMessage command is set, but no conference exists.
- “collect timeout”— The router times out waiting for digits to be keyed in by the caller.

Blocking or Non Blocking

Blocking

Example

```
# Get the destination number
set amt [getVariable radius creditAmount]
set param(interrupt) true
set param(abortKey) *
set param(dialPlan) true
set param(terminationKey) #
set ev [playPrompt param info en_you_have.au %a$amt %s1000 en_enter_dest.au]
if {$ev == "collect success"} {
   set destination $info(digits)
   set state second_authorize
   return 0
    }
    if {$ev == "collect aborted"} {
       set state get_dest
       return 0
    }
```

Usage Notes

- If there is no incoming call, the script will exit with error information.
- Unlike promptAndCollect, playPrompt does not allow generic pattern matching. It can only match against the dialPlan.

- For dynamic prompts, which are played using the **playPrompt** command, you can use the **setLocation** and **setVariable language** commands and the CLI to determine which audio file is played.For example, let's say you enter the following at the CLI:

  ```
  (config)#call application voice myApp language 3 en
  ```

  This command associates '3' is associated with the two-character language code of "en".

  Next, you specify the following in the script:

  ```
  setVariable language 3
  setLocation tftp://keyer/en_audio/ en 0
  setLocation tftp://keyer/fr_audio/ fr 0
  ```

  These commands set the current language to 3, which we identified earlier as English, and indicate that all the audio files for English are located in tftp://keyer/en_audio/ and all the audio files for French are stored in tftp://keyer/fr_audio/.

  Then, if you want to change the language without altering the script, you can use the CLI command to associate 3 with French as follows:

  ```
  (config)#call application voice myApp language 3 fr
  ```

  The result is that when the script encounters the following command:

  ```
  playPrompt welcomeMessage
  ```

  It will use the URL specified in the setLocation command for French and the audio file played will be the one located at:

  ```
  tftp://keyer/fr_audio/welcomeMessage.au
  ```

playTone

The **playTone** command creates a call progress tone. This command sends a user-specified tone to the call leg. Because there is no capability for this over an IP leg, a VoIP call leg ignores this command. If a conference is in session, the digital signalling processor (DSP) stops sending data to the remote end while playing a tone. This command is typically used give the caller a dial-tone if the script needs to collect digits.

Syntax

playTone *tone* [**outgoing**]

Arguments

The arguments of this command are:

- *tone*—This can be one of the following: Ring, Busy, Dial, Out Of Service, Address Ack, Disconnect, Off Hook Notice, Off Hook Alert, or None. When specifying the name of a tone that contains spaces, be sure to enclose the tone name in quotes. For example

  ```
  playTone “Off Hook Alert”
  ```

- **outgoing**—(Optional) If specified, the tone is played to the outgoing call leg.

Return Values

None

Blocking or Non Blocking

Non blocking

Example

```
# Output a ring for 5 seconds
playTone Ring
startTimer 5 elapsedTime
playTone None
```

Usage Notes

- If you invoke **playTone** *tone* when there is no incoming call leg, the script will exit with error information.
- If you invoke **playTone** *tone* **outgoing** when there is no outgoing call leg, the command will be ignored.

For information about how tones are defined in the code, see the “Tones” on page 62.

promptAndCollect

The **promptAndCollect** command plays a prompt to the caller and collects digits against a set of patterns. This command is similar to the **playPrompt** command, except this command does not allow dynamic prompting.

Syntax

promptAndCollect *prompt_array_name returnInfo* [*patterns*]

Arguments

The arguments of this command are:

- *prompt_array_name*—An associative array of information that describes how to play the prompt.
  - *prompt_array_name*(**url**)—Is the URL of an audio file (in G.711 8-bit ulaw format).
  - *prompt_array_name*(**abortKey**)—Specifies a key to allow the user to abort the prompt.
  - *prompt_array_name*(**terminationKey**)—Specifies a key to terminate pattern collection.
  - *prompt_array_name*(**interrupt**)—Allows the user to begin entering digits before the prompt completes playing.
  - *prompt_array_name*(**dialPlan**)—Specifies to collect digits against the dial plan.
  - *prompt_array_name*(**playComplete**)—Specifies to return on play completion.
- *returnInfo*—The name of an associative array where information about the result is stored.
- *patterns*—(Optional) An array of patterns to collect against. For more information about patterns, see the *TCL and the TK Toolkit* by John Ousterhout (published by Addison Wesley Longman, Inc).

Return Values

This command returns “collect success,” in which case returnInfo(digits) is the string collected. If the digits were collected against a pattern, returnInfo(patternName) is the name of the pattern. If the digits were collected against a dial plan, then returnInfo(patternName) is NULL.

This command returns “play complete” if prompt(playCompete) was set, or “collect fail,” if the digits collected do not match the pattern. If the digits collected do not match the pattern, the collected digits are not returned to the script.

This command returns “play fail” if the message fails to play.

If no pattern is defined and neither dialPlan nor playComplete is set to True, “collect fail” will be returned. If you use the argument “set patterns (anypattern) .+” to set the patterns, the PromptAndCollect command accepts any pattern of digits.

Blocking or Non Blocking

Blocking

Example

```
# Prompt for destination number allow * entry to abort
set prompt(url) flash:EnterDest.au
set prompt(interrupt) true
set prompt(abortKey) *
set prompt(dialPlan) true

set event [promptAndCollect prompt info]

    if {$event == “collect success”} {
       set state place_call
       set destination $info(digits)
}
```

Usage Notes

- If there is no incoming call, the script will exit with error information.
- You can collect against both the dial plan (set of dial-peers) and patterns. If a number matches both, the result is undefined.
- You cannot collect digits while a prompt is playing. Setting prompt(interrupt) allows the prompt to be terminated when the caller enters a digit.

The following arguments are defined in the code for the **promptAndCollect** command:

- PROMPT_ABORT_KEY—“abortKey”
- PROMPT_TERM_KEY—“terminationKey”
- PROMPT_URL—“url”
- PROMPT_ALLOW_INT—“interrupt”
- PROMPT_OUTGOING—“outgoing”
- PROMPT_DIALPLAN—“dialPlan”
- PROMPT_DIALPLAN_TERM—“dialPlanTerm”
- PROMPT_PLAY_RET—“playComplete”
- PROMPT_INSRTMSG—“insertMessage”
- PROMPT_MAX_DIGIT—“maxDigits”

puts

The **puts** command outputs a debug string to the console if the IVR state debug flag is set (using the **debug voip ivr state** command).

Syntax

puts [**-nonewline**] *string*

Arguments

The arguments of this command are:

- **-nonewline**—(Optional) If specified, a new line will not be output after the string.
- *string*—The string to output.

Return Values

This command returns an empty string.

Blocking or Non Blocking

Non blocking

Example:

```
puts “Handling call ID [callID]”
```

rdn

The **rdn** command returns the redirect number that came with the call.

Syntax

rdn

Arguments

None

Return Values

This command returns the redirect number or a null string if no redirect number is available.

Blocking or Non Blocking

Non blocking

Example

```
# place the call to the rdn
set event [placeCall [rdn] callInfo info]
```

Usage Notes

If there is no incoming call, the script will exit with error information.

setLocation

The **setLocation** command specifies the location of the audio files for each language and category.

Syntax

setLocation *url language category*

Arguments

The arguments of this command are:

- *url*—The location where the prompts are stored.
- *language*—If information for different languages is stored in different places, this is the two-character language code for the specified URL. Use “aa” if the prompts for all languages are stored in the same place. You can use any two characters that you want. IOS does not enforce any specific two-character identifier. So although “en” is typically used to indicate English. You could use “en” to indicate French. The two-character code should match one of the codes specified in the **call application voice** command.
- *category*—If information for different categories is stored in different places, this is the category for the specified URL. Use one of the following:
  - 0 - all
  - 1 - numbers
  - 2 - units
  - 3 - day/months
  - 4 - others

Return Values

None

Blocking or Non Blocking

Non blocking

Example

```
# install params
set accountLen [getVariable config uidLen]
set pinLen [getVariable config pinLen]
set retryCnt [getVariable config retryCount]
set operatorNum [getVariable config redirectNumber]
set warnTime [getVariable config warningTime]
set preLanguage [getVariable radius preferredLang]
setVariable language 1
setLocation tftp://keyer/echeng/bowie_audio/ en 0
```

Usage Notes

If you specify an invalid category or a language that has not been configured, the script returns an error code, which causes the underlying infrastructure to disconnect the call and clean up.

setTimeout

The **setTimeout** command sets the initial digit collection timeout and the inter-digit collection timeout values. This command overrides any configuration of the voice port. If you set the timeout to 0, this disables the timer.

Syntax

setTimeout *initialdigitTimeout interdigitTimeout*

Arguments

The arguments of this command are:

- *initialdigitTimeout*—The number of seconds to wait for the first digit in promptAndCollect.
- *interdigitTimeout*—The number of seconds to wait between digits in promptAndCollect.

Return Values

None

Blocking or Non Blocking

Non blocking

Example

```
# set the digit timeouts short for redialer interaction
setTimeout 2 1
```

setVariable

The **setVariable** command sets the value of a specified variable.

Syntax

setVariable *name value*

Arguments

The arguments of this command are:

- *name*—The name of the variable to set. Valid values are longPound and language.
- *value*—The new value. Valid values depend on the variable name.

Name	Value	Description
longPound	true, false	If longPound is false, any pound sign (#) is reported as a # digits. If longPound is true, # digits longer than one second are reported as long pound events.
language	0 through 9	Specifies the language to be used.

Return Values

If language is specified as the variable, this command can return one of the following:

- If the corresponding language has not been configured (using the **call application voice** command), this command returns a "language not specified in configuration" event and the setting is ignored.
- If the language argument is outside the range (0-9), this command returns "language not supported".
- Otherwise, this command returns a NULL string.

Blocking or Non Blocking

Non blocking

Example

```
#install parameters
setVariable language 1
setVariable longPound true
```

Usage Notes

- If a name other than longPound or language is specified, the script returns an error code, which causes the underlying infrastructure to disconnect the call and clean up.
- If longPound returns anything other than true or false, the script returns an error code, which causes the underlying infrastructure to disconnect the call and clean up.
- If a value greater than 9 is specified, a "language not supported" event is returned, but the script will not exit with error information.

setupAck

The **setupAck** command sends a call setup acknowledgement message to the incoming call leg. This command is ignored if a setup acknowledgement, proceeding, or connect message has already been sent (for example, if the script has already issued a **callProceeding** or **acceptCall** command).

Syntax

setupAck

Arguments

None

Return Values

None

Blocking or Non Blocking

Non blocking

Example

```
# acknowledge the setup before waiting on authentication
setupAck
set event [authenticate [ani] [dnis] info]
```

Usage Notes

If there is no incoming call, the script will exit with error information.

startTimer

The **startTimer** command blocks until either a specified number of seconds elapses or another event (such as a destination hang up) occurs. The timer does not continue running after the **startTimer** command returns.

Syntax

startTimer *numSeconds returnInfo*

Arguments

The arguments of this command are:

- *numSeconds*—The number of seconds to wait. Possible values are 0 to 4 million seconds.
- *returnInfo*—The name of an associative array where information about the result is stored.

Return Values

This command returns “timeout” if no other event occurs. If an event occurs, returnInfo(timeLeft) contains the amount of time left. Otherwise, it contains 0.

Blocking or Non Blocking

Blocking

Example

```
# wait for the credit time we have
set event [startTimer [creditTimeLeft] returnInfo]
while { $event == “digit” } {
   set event [startTimer [creditTimeLeft] returnInfo]
}
```

Usage Notes

- If the value for startTimer *numSeconds* is not given a number, but by mistake some alphanumeric string, the script returns an error code, which causes the underlying infrastructure to disconnect the call and clean up.
- If you specify a negative number of seconds to wait, the script returns an error code, which causes the underlying infrastructure to disconnect the call and clean up.
- If IVR state debugging is on (see the “Testing and Debugging Your Script” on page 13), this command will cause “Wait for <num> seconds” to be displayed when this command is encountered.

waitEvent

The **waitEvent** command causes the script to sit idly until an event occurs.

Syntax

waitEvent *returnInfo*

Arguments

The argument of this command is:

- *returnInfo*—The name of an associative array where information about the result is stored.

Return Values

This command returns:

- "outgoing disconnected" if the outgoing call leg disconnected.
- "digit" if a digit was collected. In this case, returnInfo(digits) will contain the digit collected.

Blocking or Non Blocking

Blocking

Example

```
# Place a call, then just sit on waitEvent, ignoring digits
set callInfo(destinationNum) destination
set event [placeCall destination callInfo info]

if {$event != active} {
    puts "Call [callID] got $event when placing call"
    exit 0
}

set event [waitEvent returnInfo]
while {$event == "digit" } {
   puts "ignoring received digit $returnInfo(digits)"
   set event [waitEvent]
}
```

Value Definitions

Within the code that supports the TCL IVR scripts, there is a series of defined values. These values are listed in this section.

Events

The valid event values for the TCL IVR commands are:

- TA_E_AUTHEN_SUCCESS—“authenticated”
- TA_E_AUTHEN_FAIL—“authentication failed”
- TA_E_AUTHOR_SUCCESS—“authorized”
- TA_E_AUTHOR_FAIL—“authorization failed”
- TA_E_AUTHOR_ERROR—“authorization error”
- TA_E_DISC_OUT—“outgoing disconnected”
- TA_E_CALL_FAIL —“call fail”
- TA_E_ACTIVE—“active”
- TA_E_COLLECT_FAIL—“collect fail”
- TA_E_COLLECT_SUCCESS—“collect success”
- TA_E_COLLECT_COMPLETE—“play complete”
- TA_E_COLLECT_ABORTED—“collect aborted”
- TA_E_COLLECT_TIMEOUT—“collect timeout”
- TA_E_PLAY_COMPLETE—“play complete”
- TA_E_CANT_PLAY_PROMPT—“play failed on voip call leg”
- TA_E_FAILURE—“failure”
- TA_E_SUCCESS—“success”
- TA_E_DIGIT—“digit”
- TA_E_TIMEOUT—“timeout”
- TA_E_CREATE_DONE—“conference created”
- TA_E_DESTROY_DONE—“conference deleted”
- TA_E_LONG_POUND—“longpound”
- TA_E_FACILITY—“facility”

ReturnInfo Array

The retInfo variable in most commands is an associative array that can contains information in the following fields:

- TA_RI_CODE—“code”
- TA_RI_DIGITS—“digits”
- TA_RI_PATNAME—“patternName”
- TA_RI_CREDIT—“creditTime”
- TA_RI_TIMELEFT—“timeLeft”
- TA_RI_LONG_POUND—“longPound”
- TA_RI_ROAM—“roam”

Tones

For tones, the possible values are:

- TA_TONE_RINGBACK—“Ring”
- TA_TONE_BUSY—“Busy”
- TA_TONE_DIALTONE—“Dial”
- TA_TONE_OOS—“Out Of Service”
- TA_TONE_ADDR_ACK—“Address Ack”
- TA_TONE_DISCONNECT—“Disconnect”
- TA_TONE_OFF_HOOK_NOTICE—“Off Hook Notice”
- TA_TONE_OFF_HOOK_ALERT—“Off Hook Alert”
- TA_TONE_NULL—“None”

Related Documentation

- Configuring Interactive Voice Response for Cisco Access Platforms

  http://www.cisco.com/univercd/cc/td/doc/product/access/acs_serv/5300/cfios/cfselfea/0061ivr.htm

- Service Provider Features for Voice over IP

  http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120t/120t3/voip1203.htm

- Voice over IP for the Cisco AS5300

  http://www.cisco.com/univercd/cc/td/doc/product/access/nubuvoip/voip5300/index.htm

- Voice over IP for the Cisco AS5800

  http://www.cisco.com/univercd/cc/td/doc/product/access/nubuvoip/voip5800/index.htm

- Voice over IP for the Cisco 2600/Cisco 3600 Series

  http://www.cisco.com/univercd/cc/td/doc/product/access/nubuvoip/voip3600/index.htm

- Configuring H.323 VoIP Gateway for Cisco Access Platforms

  http://www.cisco.com/univercd/cc/td/doc/product/access/acs_serv/5300/iosinfo/ios_mods/0044gw.htm

- Configuring H.323 VoIP Gatekeeper for Cisco Access Platforms

  http://www.cisco.com/univercd/cc/td/doc/product/access/acs_serv/5300/iosinfo/ios_mods/0042gk.htm

- Prepaid Distributed Calling Card via Packet Telephony

  http://www.cisco.com/univercd/cc/td/doc/product/access/acs_serv/5300/cfios/cfselfea/0134bowi.htm

- RADIUS Vendor-Specific Attributes Voice Implementation Guide

  http://www.cisco.com/univercd/cc/td/doc/product/access/acs_serv/vapp_dev/vsaig.htm

Glossary

AAA—Authentication, Authorization, and Accounting. A suite of network security services that provides the primary framework through which you can set up access control on your Cisco router or access server.

ANI—Automatic number identification. Same as calling party.

CDR—Call Data Record.

CLI—Command line interface.

DID—Dial in direct. Calls in which the gateway uses the number that you initially dialed (DNIS) to make the call, as opposed to prompting you to dial additional digits.

DNIS—Dialed number identification service. Same as the called number.

DSP—Digital signalling processor.

DTMF—Dual tone multifrequency. Use of two simultaneous voice-band tones for dialing (such as touch tone).

Gatekeeper—A Gatekeeper maintains a registry of devices in the multimedia network. The devices register with the Gatekeeper at startup and request admission to a call from the Gatekeeper.

The Gatekeeper is an H.323 entity on the LAN that provides address translation and control access to the LAN for H.323 terminals and gateways. The Gatekeeper can provide other services to the H.323 terminals and gateways, such as bandwidth management and locating gateways.

gateway—A gateway allows H.323 terminals to communicate with non-H.323 terminals by converting protocols. A gateway is the point where a circuit-switched call is encoded and repackaged into IP packets.

An H.323 gateway is an endpoint on the LAN that provides real-time, two-way communications between H.323 terminals on the LAN and other ITU-T terminals in the WAN or to another H.323 gateway.

IFS—Cisco IOS File System.

ISDN—Integrated Services Digital Network. Communication protocol, offered by telephone companies, that permits telephone networks to carry data, voice, and other source traffic.

IVR—Interactive voice response. When someone dials in, IVR responds with a prompt to get a personal identification number (PIN), and so on.

OSP—Open Settlement Protocol. Protocol used to allow carriers to communicate with third-party settlement servers.

PIN—Personal identification number. Password used with account number for authentication.

POTS—Plain old telephone service. Basic telephone service supplying standard single line telephones, telephone lines, and access to the PSTN.

PSTN—Public Switched Telephone Network. PSTN refers to the local telephone company.

RADIUS—Remote Access Dial-In User Service. Database for authenticating modem and ISDN connections and for tracking connection time.

TCL—Tool Command Language. TCL is an interpreted script language developed by Dr. John Ousterhout of the University of California, Berkeley, and is now developed and maintained by Sun Microsystems Laboratories.

URL—Universal Resource Locator. Standardized addressing scheme for accessing hypertext documents and other services using a browser.

VoFR—Voice Over Frame Relay. Voice over Frame Relay enables a router to carry voice traffic (for example, telephone calls and faxes) over a Frame Relay network. When sending voice traffic over Frame Relay, the voice traffic is segmented and encapsulated for transit across the Frame Relay network using FRF.12 encapsulation.

VoIP—Voice over IP. The ability to carry normal telephone-style voice signals over an IP-based network with POTS-like functionality, reliability, and voice quality. VoIP is a blanket term that generally refers to Cisco's open standards-based (for example, H.323) approach to IP voice traffic.

Note For a list of other internetworking terms, see Internetworking Terms and Acronyms document that is available on the Documentation CD-ROM and Cisco Connection Online (CCO) at the following URL: http://www.cisco.com/univercd/cc/td/doc/cisintwk/ita/index.htm.

Cisco Connection Online

Cisco Connection Online (CCO) is Cisco Systems' primary, real-time support channel. Maintenance customers and partners can self-register on CCO to obtain additional information and services.

Available 24 hours a day, 7 days a week, CCO provides a wealth of standard and value-added services to Cisco's customers and business partners. CCO services include product information, product documentation, software updates, release notes, technical tips, the Bug Navigator, configuration notes, brochures, descriptions of service offerings, and download access to public and authorized files.

CCO serves a wide variety of users through two interfaces that are updated and enhanced simultaneously: a character-based version and a multimedia version that resides on the World Wide Web (WWW). The character-based CCO supports Zmodem, Kermit, Xmodem, FTP, and Internet e-mail, and it is excellent for quick access to information over lower bandwidths. The WWW version of CCO provides richly formatted documents with photographs, figures, graphics, and video, as well as hyperlinks to related information.

You can access CCO in the following ways:

- WWW: http://www.cisco.com
- WWW: http://www-europe.cisco.com
- WWW: http://www-china.cisco.com
- Telnet: cco.cisco.com
- Modem: From North America, 408 526-8070; from Europe, 33 1 64 46 40 82. Use the following terminal settings: VT100 emulation; databits: 8; parity: none; stop bits: 1; and connection rates up to 28.8 kbps.

For a copy of CCO's Frequently Asked Questions (FAQ), contact cco-help@cisco.com. For additional information, contact cco-team@cisco.com.

Note If you are a network administrator and need personal technical assistance with a Cisco product that is under warranty or covered by a maintenance contract, contact Cisco's Technical Assistance Center (TAC) at 800 553-2447, 408 526-7209, or tac@cisco.com. To obtain general information about Cisco Systems, Cisco products, or upgrades, contact 800 553-6387, 408 526-7208, or cs-rep@cisco.com.

Documentation CD-ROM

Cisco documentation and additional literature are available in a CD-ROM package, which ships with your product. The Documentation CD-ROM, a member of the Cisco Connection Family, is updated monthly. Therefore, it might be more current than printed documentation. To order additional copies of the Documentation CD-ROM, contact your local sales representative or call customer service. The CD-ROM package is available as a single package or as an annual subscription. You can also access Cisco documentation on the World Wide Web at http://www.cisco.com, http://www-china.cisco.com, or http://www-europe.cisco.com.

If you are reading Cisco product documentation on the World Wide Web, you can submit comments electronically. Click **Feedback** in the toolbar and select **Documentation**. After you complete the form, click **Submit** to send it to Cisco. We appreciate your comments.

This document is to be used in conjunction with the documents listed in the “Related Documentation” section.