Using Tcl IVR Scripts

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

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

How Tcl IVR Version 2.0 Works

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

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

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

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

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

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

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

  The FSM supports two wildcard states and one wildcard event:

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

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

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

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

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

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

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

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

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

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

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

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

Writing an IVR Script Using Tcl Extensions

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

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

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

Prompts in Tcl IVR Scripts

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

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

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

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

Sample Tcl IVR Script

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

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

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

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

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

Next, we define a series of procedures.

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

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

```
proc init { } {
    global param

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

}
```

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

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

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

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

    set beep 0
    leg setupack leg_incoming

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

        playtone leg_incoming tn_dial

        set param(dialPlan) true
        leg collectdigits leg_incoming param
      }

}
```

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

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

```
proc act_GotDest { } {
    global dest

    set status [infotag get evt_status]

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

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

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

- When the call is successful (ls_000), the script obtains the amount of credit time.
- If a value other than unlimited or uninitialized is returned, a timer is started.
- If the call is not successful, the script reports the error and closes the call. For a list of other leg setup status values, see the "Leg Setup Status" section on page 5-11.

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

```
    set status [infotag get evt_status]

    if { $status == "ls_000"} {

        set  creditTimeLeft [infotag get leg_settlement_time leg_outgoing]

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

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

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

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

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

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

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

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

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

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

    connection create $incoming $outgoing
}
```

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

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

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

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

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

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

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

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

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

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

```
init

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

Initialization and Setup of State Machine

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

```
fsm define TopFSM  CALL_INIT
```

Testing and Debugging Your Script

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

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

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

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

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

Possible sources of errors are:

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

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

Loading Your Script

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

```
Router# call application voice application_name script_url
```

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

```
Router# call application voice application_name script_url [parameter value]
```

In this command:

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

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

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

```
Router# call application voice test tftp://keyer/debit_audio/newapp.tcl
```

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

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

Associating Your Script with an Inbound Dial Peer

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

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

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

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

In these commands:

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

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

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

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

Displaying Information About IVR Scripts

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

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

In this command:

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

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

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

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

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

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


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


proc init { } {
    global param

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

}

proc act_Setup { } {
    global dest
    global beep

    set beep 0
    leg setupack leg_incoming

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

        playtone leg_incoming tn_dial

        set param(dialPlan) true
        leg collectdigits leg_incoming param
      }


}

proc act_GotDest { } {
    global dest

    set status [infotag get evt_status]

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

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

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


proc act_CallSetupDone { } {
    global beep

    set status [infotag get evt_status]

    if { $status == "CS_000"} {

        set  creditTimeLeft [infotag get leg_settlement_time leg_outgoing]

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

proc act_Timer { } {
    global beep
    global incoming
    global outgoing

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

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

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

proc act_Beeped { } {
    global incoming
```

```
    global outgoing

    connection create $incoming $outgoing
}

proc act_ConnectedAgain { } {
    timer start leg_timer 10 leg_incoming
}

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

proc act_Cleanup { } {
    call close
}

init

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


  fsm define TopFSM  CALL_INIT
```

Using URLs in IVR Scripts

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

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

URLs for Loading the IVR Script

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

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

URLs for Loading Audio Files

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

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

Tips for Using Your Tcl IVR Script

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

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

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

- What happens if my script encounters an error?

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

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