# APPENDIX B
# Application Reference

## AbsoluteTimeout( )

Sets the maximum number of seconds a call may last

```
AbsoluteTimeout(length)
```

Sets the absolute time limit of a call to *length* seconds. Calls lasting longer than *length* seconds will be sent to the T (absolute timeout) extension, if it exists. Otherwise, the channel will be hung up.

If *length* is set to zero (0), the timeout is disabled.

Each time AbsoluteTimeout( ) runs, it overrides the previous timeout setting. Asterisk starts the timeout countdown at the time the application is called, not at the time the call starts.

```
; limit calls to ex-girlfriend to 300 seconds
exten => 123,1,AbsoluteTimeout(300)
exten => 123,2,Dial(${EX-GIRLFRIEND})
exten => T,1,Playback(im-sorry)
exten => T,2,Playback(vm-goodbye)
exten => T,3,Hangup( )
```

### See Also

DigitTimeout( ), ResponseTimeout( ), the T extension

## AddQueueMember( )

Dynamically adds queue members to the specified call queue

```
AddQueueMember(queuename[,interface[,penalty]])
```

Dynamically adds the specified *interface* to an existing queue named *queuename*, as specified in *queues.conf*. If specified, *penalty* sets the penalty for queues to use this member. Members with a lower penalty are called before members with a higher penalty.

If *interface* is already a member of the queue and there exists an n+101 priority (where n is the number of the current priority), the call will continue at that priority. Otherwise, it will return an error.

Calling AddQueueMember( ) without an *interface* argument will use the interface that the caller is currently using.

```
; add SIP/3000 to the techsupport queue, with a penalty of 1
exten => 123,1,AddQueueMember(techsupport,SIP/3000,1)
```

### See Also

RemoveQueueMember( ), *queues.conf*

## ADSIProg( )

Loads an ADSI script into an ADSI-capable phone

ADSIProg(*script*)

Programs an Analog Display Services Interface (ADSI) phone with the given *script*. If none is specified, the default script, *asterisk.adsi*, is used. The path for the *script* is relative to the Asterisk configuration directory (usually */etc/asterisk/*). You may also provide the full path to the script.

To get the CPE ID and other information from your ADSI-capable phone, use the GetCPEID( ) application.

```
; program the ADSI phone with the telcordia-1.adsi script
exten => 123,1,ADSIProg(telcordia-1.adsi)
```

### See Also

GetCPEID( ), *adsi.conf*

## AgentCallbackLogin( )

Enables agent login with callback

AgentCallbackLogin([*AgentNo*][,[*options*][*exten*]@*context*])

Allows a call agent identified by *AgentNo* to log into the call queue system, to be called back when a call comes in for that agent.

When a call comes in for the agent, Asterisk calls the specified *exten* (with an optional *context*).

The *options* argument may contain the letter s, which causes the login to be silent.

```
; silently log in as agent number 42, and have Asterisk
; call SIP/400 when a call comes in for this agent
exten => 123,1,AgentCallbackLogin(42,s,SIP/400)
```

### See Also

AgentLogin( )

## AgentLogin( )

Allows a call agent to log into the system

AgentLogin([*AgentNo*][,*options*])

Logs the current caller into the call queue system as a call agent (optionally identified by *AgentNo*). While logged in, the agent can receive calls and will hear a beep on the line when a new call comes in. The agent can hang up the call by pressing the asterisk (*) key.

The *options* argument may contain the letter s, which causes the login to be silent.

```
; silently log in as agent number 42, as defined in agents.conf
exten => 123,1,AgentLogin(42,s)
```

### See Also

AgentCallbackLogin( )

## AgentMonitorOutgoing( )

Records an agent's outgoing calls

AgentMonitorOutgoing([*options*])

Records all outbound calls made by a call agent.

This application tries to figure out the ID of the agent who is placing outgoing call based on a comparison of the Caller ID of the current interface and the global variable set by the AgentCallbackLogin( ) application. As such, it should be used only in conjunction with (and after!) the AgentCallbackLogin( ) application. It uses the monitoring functions in the chan_agent module instead of the Monitor( ) application to record the calls. This means that call recording must be configured correctly in the *agents.conf* file.

By default, recorded calls are saved to the */var/spool/asterisk/monitor/* directory. This may be overridden by changing the savecallsin parameter in *agents.conf*.

If the Caller ID and/or agent ID are not found, this application will go to priority n+1, if it exists (where n is the current priority).

Returns 0 unless overridden by one of the options.

The *options* argument may include one or more of the following:

d Make this application return -1 if there is an error condition and there is no extension n+101.

c Change the Call Detail Record so that the source of the call is recorded as Agent/agent_id.

n Don't generate warnings when there is no Caller ID or if the agent ID is not known. This option is useful if you want to have a shared context for agent and non-agent calls.

```
; record outbound calls for this agent, and change the CDR to reflect
; that the call is being made by an agent
exten => 123,1,AgentMonitorOutgoing(c)
```

### See Also

AgentCallbackLogin( ), *agents.conf*

## AGI( )

Executes an AGI-compliant application

[E]AGI(*program*[,*arguments*])

Executes an Asterisk Gateway Interface–compliant *program* on the current channel. AGI programs allow external programs (written in almost any language) to control the telephony channel by playing audio, reading DTMF digits, and so on. Asterisk communicates

with the AGI program on STDIN and STDOUT. The specified *arguments* are passed to the AGI program.

The *program* must be set as executable in the underlying filesystem. The program path is relative to the Asterisk AGI directory, which by default is */var/lib/asterisk/agi-bin/*.

If you want to run an AGI when no channel exists (such as in an h extension), use the DeadAGI( ) application instead. You may want to use the FastAGI( ) application if you want to do AGI processing across the network.

If you want access to the inbound audio stream from within your AGI program, use EAGI( ) instead of AGI( ). Inbound audio can then be read in on file descriptor number three.

Returns -1 on hangup or if the program requested a hangup, or 0 on non-hangup exit.

```
; call the demo AGI program
exten => 123,1,AGI(agi-test)
exten => 123,2,EAGI(eagi-test)
```

### See Also

DeadAGI( ), FastAGI( ), Chapter 9

## AlarmReceiver( )

Provides support for receiving alarm reports from a burglar or fire alarm panel

AlarmReceiver( )

Emulates an alarm receiver, and allows Asterisk to receive and decode special data from fire and/or burglar alarm panels. At this time, only the Ademco Contact ID format is supported.

When called, AlarmReceiver( ) will handshake with the alarm panel, receive events, validate them, handshake them, and store them until the panel hangs up. Once the panel hangs up, the application will run the command line specified by the eventcmd setting in *alarmreceiver.conf* and pipe the events to the standard input of the application. *alarmreceiver.conf* also contains settings for DTMF timing and for the loudness of the acknowledgment tones.

This application is not guaranteed to be reliable, so don't depend on it unless you have extensively tested it. If you use this application without extensive testing, you may be putting your life and property at great risk.

This application always returns 0.

```
; set up Asterisk to answer a call from a supported fire alarm panel
exten => s,1,AlarmReceiver( )
```

### See Also

*alarmreceiver.conf*

## Answer( )

Answers a channel, if it is ringing

```
Answer( )
```

Causes Asterisk to answer the channel if it is currently ringing. If the current channel is not ringing, this application does nothing.

It is usually a good idea to use `Answer( )` on the channel before calling any other applications, unless you have a very good reason not to. Most applications require that the channel be answered before they are called, and may not work correctly otherwise.

Returns `0` unless it tries to answer the channel and fails.

```
exten => 123,1,Answer( )
exten => 123,2,Wait(1)
exten => 123,3,Playback(tt-weasels)
```

### See Also

`Hangup( )`

## AppendCDRUserField( )

Appends a value to the user field of the Call Detail Record

```
AppendCDRUserField(value)
```

Appends *value* to the user field of the Call Detail Record (CDR). The user field is often used to store arbitrary data about the call, which may not be appropriate for any of the other fields.

Always returns `0`.

```
; set the user field to 'abcde'
exten => 123,1,SetCDRUserField(abcde)
; now append 'xyz'
exten => 123,1,AppendCDRUserField(xyz)
```

### See Also

`SetCDRUserField( )`, `ForkCDR( )`, `NoCDR( )`, `ResetCDR( )`

## Authenticate( )

Requires that the caller enter a correct password before continuing

```
Authenticate(password[,options])
```

Requires a caller to enter a given *password* in order to continue execution of the next priority in the dialplan. `Authenticate( )` gives the caller three chances to enter the password correctly. If the password is not correctly entered after three tries, the channel is hung up.

If *password* begins with the `/` character, it is interpreted as a file that contains a list of valid passwords (one per line). Passwords may also be stored in the Asterisk database (AstDB); see the `d` option below.

A set of *options* may be provided, consisting of one or more of the letters in the following list.

a Sets the CDR field named accountcode and the channel variable ACCOUNTCODE to the password that is entered

d Interprets the path as the database key from the Asterisk database in which to find the password, not a literal file. When using a database key, the value associated with the key can be anything.

r Removes the database key upon successful entry (valid with d only).

Returns 0 if the user enters a valid password within three tries, or -1 otherwise (or on hangup).

```
; force the caller to enter the password before continuing, and set the CDR field
; named 'accountcode' to the entered password
exten => 123,1,Answer
exten => 123,2,Authenticate(1234,a)
exten => 123,3,Playback(pin-number-accepted)
exten => 123,4,SayDigits(${ACCOUNTCODE})
```

### See Also

VMAuthenticate( ), Chapter 6

## Background( )

Plays a file while accepting touch-tone (DTMF) digits

Background(*filename1*[&*filename2*...][,*options*[,*language*]])

Plays the specified audio file(s) while waiting for the user to begin entering an extension. Once the user begins to enter an extension, the playback is terminated. The *filename* should be specified without a file extension, as Asterisk will automatically find the file format with the lowest translation cost.

Valid *options* include one of the following:

skip
: Causes the playback of the message to be skipped if the channel is not in the "up" state (i.e., hasn't yet been answered). If skip is specified, the application will return immediately should the channel not be off-hook.

noanswer
: Does not answer the channel before playing the specified file. Without this option, the channel will automatically be answered before the sound is played. Not all channels support playing messages before being answered.

The *language* argument may be used to specify a language to use for playing the prompt, if it differs from the current language of the channel.

Returns -1 if the channel was hung up, or if the given filename does not exist; otherwise, returns 0.

```
exten => 123,1,Answer( )
exten => 123,2,Background('exter-ext-of-person');
```

### See Also

Playback( ), BackgroundDetect( ), the show translation command

## BackgroundDetect( )

Plays a file in the background and detects talking

BackgroundDetect(*filename*[,*sil*[,*min*[,*max*]]])

Similar to Background( ), but attempts to detect talking.

During the playback of the file, audio is monitored in the receive direction. If a period of non-silence that is greater than *min* milliseconds yet less than *max* milliseconds and is followed by silence for at least *sil* milliseconds occurs, the audio playback is aborted and processing jumps to the talk extension, if available.

If unspecified, *sil*, *min*, and *max* default to 1,000 ms, 100 ms, and infinity, respectively.

Returns -1 on hangup, and 0 on successful playback completion with no exit conditions.

```
exten => 123,1,BackgroundDetect(tt-monkeys)
exten => 123,2,Playback(im-sorry)
exten => talk,1,Playback(yes-dear)
```

### See Also

Playback( ), Background( )

## Busy( )

Indicates a busy condition to the channel

Busy([*timeout*])

Requests that the channel indicate the busy condition and then waits for the user to hang up or for the optional *timeout* (in seconds) to expire.

This application only signals a busy condition to the bridged channel. Each particular channel type has its own way of communicating the busy condition to the caller. You can use Playtones(busy) to play a busy tone to the caller.

Always returns -1.

```
exten => 123,1,Playback(im-sorry)
exten => 123,2,Playtones(busy)
exten => 123,3,Busy( )
```

### See Also

Congestion( ), Progress( ), Playtones( )

## CallingPres( )

Changes the presentation for the Caller ID

CallingPres(*presentation*)

Changes the presentation parameters for the Caller ID on a Q931 PRI connection. These parameters should be set before placing an outgoing call. The argument *presentation* controls two things: whether or not the person being called can view the Caller ID information (known as *presentation*), and whether or not the Caller ID information has been verified by an authoritative source (known as *screening*).

This application has been replaced by the SetCallerPres( ) application, which is easier to use and less dependent on the internal Zaptel structures.

This application takes the call presentation setting and the screening setting and combines them into one number. The values themselves are defined in the ITU Q931 standard, as shown in Tables B-1 and B-2.

*Table B-1. Screening is controlled by bits 2 and 1*

| Bit 2 | Bit 1 | Explanation |
|---|---|---|
| 0 | 0 | Caller ID information was provided by the user, and not screened. |
| 0 | 1 | Caller ID information was provided by the user, and successfully verified. |
| 1 | 0 | Caller ID information was provided by the user, and verification failed. |
| 1 | 1 | Caller ID information was provided by the network. |

*Table B-2. Presentation is controlled by bits 7 and 6*

| Bit 7 | Bit 6 | Explanation |
|---|---|---|
| 0 | 0 | Presentation of the Caller ID information is allowed. |
| 0 | 1 | Presentation of the Caller ID information is restricted. |
| 1 | 0 | The number is not available due to interworking. |
| 1 | 1 | Reserved. |

Bits 3, 4, 5, and 8 should all be set to zero (0). Please note that the bits are numbered from most significant to least significant, like this: 87654321.

```
; set presentation to:
; Presentation Allowed                (00000000)
; Network Provided                    (00000011)
; ------------------                  ----------
; Result = 3 (bitwise AND)            (00000011)
exten => 123,1,CallingPres(3)
exten => 123,2,Dial(Zap/g1/8885551212)

; set presentation to:
; Presentation Restricted             (00100000)
; User-provided, verified, and passed (00000001)
; ------------------                  ----------
; Result = 33 (bitwise AND)           (00100001)
exten => 124,1,CallingPres(33)
exten => 124,2,Dial(Zap/g1/8885551213)
```

## See Also

SetCallerPres( ), SetCallerID( )

## ChangeMonitor( )

Changes the monitoring filename of a channel

ChangeMonitor(*filename_base*)

Changes the name of the recorded file created by monitoring a channel with the Monitor( ) application. This application has no effect if the channel is not monitored. The argument *filename_base* is the new filename base to use for monitoring the channel.

```
; start recording this channel with a basename of 'sample'
exten => 123,1,Monitor(sample)
; change the filename base to 'example'
exten => 123,2,ChangeMonitor(example)
```

### See Also

Monitor( ), StopMonitor( )

## ChanIsAvail( )

Finds out if a specified channel is currently available

ChanIsAvail(*technology1/resource1*[&*technology2/resource2*...][,*option*])

Checks to see if any of the requested channels are available. If none of the requested channels are available, the new priority will be n+101 (where n is the current priority), unless that priority does not exist or an error occurs, in which case ChanIsAvail( ) will exit and return -1.

If any of the requested channels are available, the next priority will be n+1 and ChanIsAvail( ) will return 0.

ChanIsAvail( ) sets the following channel variables:

${AVAILCHAN}
: The name of the available channel, including the call session number used to perform the test.

${AVAILORIGCHAN}
: The canonical channel name that was used to create the channel—that is, the channel name without any session number.

${AVAILSTATUS}
: The status code for the channel.

If the option s (which stands for "state") is specified, Asterisk will consider the channel unavailable whenever it is in use, even if it can take another call.

This application does not work correctly on MGCP channels.

```
; check both Zap/1 and Zap/2 to see if they're available
exten => 123,1,ChanIsAvail(Zap/1&Zap/2)
; if we go to priority 2, then one of the channels is available
; in priority 2, we'll dial our number on the available channel
exten => 123,2,NoOp(${AVAILORIGCHAN})
```

```
exten => 123,3,Dial(${AVAILORIGCHAN}/5551212)
; if we go to priority 101, then neither Zap/1 nor Zap/2 is available
exten => 123,3,Playback(all-circuits-busy-now)
```

## CheckGroup( )

Checks the number of channels in a particular group

CheckGroup(*max*[@*category*])

Checks to see if the total number of channels in the current channel's group exceeds the *max* argument. If the number does not exceed *max*, the application continues to the next priority. If the number of channels in the group is higher than *max*, and priority n+101 exists (where n is the current priority), execution continues at that priority. Otherwise, the application terminates and -1 is returned.

When the optional *category* argument is passed, this application checks the total number of channels in the group category. See SetGroup( ) for more information about categories.

```
exten => 123,1,SetGroup(support)
exten => 123,2,CheckGroup(5)
; if there are less than five calls in the support group
exten => 123,3,Dial(${SUPPORT})
; if there are more than five calls in the support group
exten => 123,103,Playback(im-sorry)
```

### See Also

SetGroup( ), GetGroupCount( ), GetGroupMatchCount( )

## Congestion( )

Indicates congestion on the channel

Congestion([*timeout*])

Requests that the channel indicate congestion and then waits for the user to hang up or for the optional *timeout* (in seconds) to expire.

This application only signals congestion; it doesn't actually play a congestion tone to the user. You can use Playtones(congestion) to play a congestion tone to the caller.

Always returns -1.

```
; if the Caller ID is 555-1234, always play congestion
exten => 123,1,GotoIf($[${CALLERIDNUM} = 5551234]?5:2)
exten => 123,2,Playtones(congestion)
exten => 123,3,Congestion( )
exten => 123,4,Hangup( )
exten => 123,5,Dial(Zap/1)
```

### See Also

Busy( ), Progress( ), Playtones( )

## ControlPlayback( )

Plays a file, with the ability to fast forward and rewind the file

ControlPlayback(*filename*[,*skipms*[,*ffchar*[,*rewchar*[,*stopchar*[,*pausechr*]]]]])

Plays back a given filename (without the file extension), while allowing the caller to move forward and backward through the file by pressing *ffchar* and *rewchar*. By default, you can use * and # to rewind and fast-forward the playback of the file. If *stopchar* is specified, the application will stop playback when *stopchar* is pressed. If the file does not exist, the application jumps to priority n+101, if present (where n is the current priority number).

The *skipms* option specifies how far forward or backward to jump in the file with each press of *ffchar* or *rewchar*.

A *pausechr* option may also be specified, which will pause playback of the file. Pressing *pausechr* again will continue the playback of the file.

Returns -1 if the channel was hung up during playback.

```
; allow the caller to control the playback of this file
exten => 123,1,ControlPlayback(tt-monkeys|3000|#|*|5|0)
```

### See Also

Playback( ), Background( )

## Curl( )

Loads an external URL and assigns the result to a variable

Curl(*URL*[,*postdata*])

Downloads the given *URL* and assigns it to the channel variable named CURL. If specified, the *postdata* argument is passed to the URL as an HTTP POST. Curl( ) is often used to signal external applications of dialplan events.

Returns 0, or -1 on fatal errors.

```
; post the Caller ID number and unique call ID to a URL
exten => 123,1,Curl(http://localhost/test.
php,CallerID=${CALLERID}&UniqueCallID={$UNIQUEID})
; now use the NoOp( ) application to print the result to the Asterisk console
exten => 123,2,NoOp(${CURL})
```

## Cut( )

Assigns part of one variable to another variable

Cut(*newvar=varname,delimiter,fieldspec*)

Cuts an existing variable named *varname* into several pieces, and assigns one or more of the pieces to a new variable named *newvar*.

The *delimiter* argument is the character on which to cut *varname*. It defaults to -.

*fieldspec* is the number of the field you want to assign to *newvar*. Fields are counted starting with 1. The *fieldspec* may be specified as a range (with -) or a group of ranges and fields (with &). If more than one field is selected, Cut( ) leaves the delimiter between the fields.

Returns 0, or -1 on hangup or error.

```
exten => 123,1,Set(TEST=123-456-7890)
exten => 123,2,Cut(FIRST=TEST,-,2)        ; gives us 456
exten => 123,3,Cut(SECOND=TEST,,1-2)      ; gives us 123-456
exten => 123,4,Cut(THIRD=TEST,-,1&3)      ; gives us 123-7890
```

## DateTime( )

Says the specified time in a custom format

DateTime([*unixtime*][,*timezone*[,*format*]])

Says the time *unixtime*, in the time zone specified by *timezone*, according to the format specified in *format*.

The *unixtime* argument is the time, in seconds, since January 1, 1970. It may be negative for dates before 1970. *unixtime* defaults to the current time.

The *timezone* argument specifies the time zone of the specified time. See */usr/share/zoneinfo/* for a list of valid time zones. *timezone* defaults to the current time zone of the Asterisk server.

The *format* argument specifies which parts of the date and time should be read. See *voicemail.conf* for formatting options. *format* defaults to "ABdY 'digits/at' IMp".

Returns 0, or -1 on hangup.

```
; today's date and time
exten => 123,1,DateTime( )
; today's date
exten => 124,1,DateTime(,,BdY)
; A specified date
exten => 125,1,DateTime(871624800,,BdY)
```

## DBdel( )

Deletes a key from the AstDB

DBdel(*family/key*)

Deletes the key specified by *key* from the key family named *family* in the AstDB.

Always returns 0.

```
exten => 123,1,DBput(test/name=John) ; add name to AstDB
exten => 123,2,DBget(NAME=test/name) ; retrieve name from AstDB
exten => 123,3,DBdel(test/name)      ; delete from AstDB
```

### See Also

DBdeltree( ), DBput( ), DBget( )

## DBdeltree( )

Deletes a family or key tree from the Asterisk database

DBdeltree(*family*[/*keytree*])

Deletes the specified *family* or *keytree* from the AstDB.

Always returns 0.

```
; create a couple of entries in the AstDB
exten => 123,1,DBput(test/blue)
exten => 123,2,DBput(test/green)
; now delete the key family named test
exten => 123,3,DBdeltree(test)
```

### See Also

DBdel( ), DBput( ), DBget( )

## DBget( )

Retrieves a key from the AstDB

DBget(*varname=family/key*)

Retrieves a key value from the Asterisk database and stores it in the variable specified by *varname*. If the requested key is not found, control jumps to priority n+101 (where n is the current priority), if it exists.

Always returns 0.

```
; put an entry in the AstDB
exten => 123,1,DBput(test/color=blue)
; now retrieve it and assign it to a variable
exten => 123,2,DBget(COLOR=test/color)
```

### See Also

DBdel( ), DBdeltree, DBput( )

## DBput( )

Stores a value in the AstDB

DBput(*family/key=value*)

Stores the given *value* in the corresponding *family* and *key* in the AstDB.

Always returns 0.

```
; put an entry in the AstDB
exten => 123,1,DBput(test/color=blue)
```

### See Also

DBdel( ), DBdeltree, DBget( )

## DeadAGI( )

Executes an AGI-compliant script on a dead (hung-up) channel

DeadAGI(*program,args*)

Executes an AGI-compliant *program* on a dead (hung-up) channel. AGI allows Asterisk to launch external programs written in almost any language to control a telephony channel, play audio, read DTMF digits, and so on by communicating with the AGI protocol on STDIN and STDOUT. The arguments specified by *args* will be passed to the program.

This application has been written specifically for dead channels, as the normal AGI interface doesn't work correctly if the channel has been hung up.

Use the `show agi` command on the command-line interface to list all of the available AGI commands.

Returns `-1` if the application requested a hangup, or `0` on a non-hangup exit.

```
exten => h,1,DeadAGI(agi-test)
```

### See Also

`AGI( )`, `FastAGI( )`

## Dial( )

Attempts to connect channels

`Dial(`*tech/username:password@hostname/extension,ring-timeout,flags*`)`

Allows you to connect together all of the various channel types.* `Dial( )` is the most important application in Asterisk—you'll want to read through this section a few times.

Any valid channel type (such as SIP, IAX2, H.323, MGCP, Local, or Zap) is acceptable to `Dial( )`, but the parameters that need to be passed to each channel will depend on the information the channel type needs to do its job. For example, a SIP channel will need a network address and user to connect to, whereas a Zap channel is going to want some sort of phone number.

When you specify a channel type that is network-based, you can pass the destination host (name or IP address), username, password, and remote extension as part of the options to `Dial( )`, or you can refer to the name of a channel entry in the appropriate *.conf* file; all the required information will then need to be obtained from that file. The username and password can be replaced with the name contained within square brackets ([]) of the channel configuration file. The hostname is optional.

This is a valid `Dial` statement:

```
exten => s,1,Dial(SIP/sake:arigato@thathostoverthere.tld)
```

This is effectively identical:

```
exten => s,1,Dial(SIP/some_SIP_friend)
```

but will work only if there is a channel defined in *sip.conf* as `[some_SIP_friend]`, whose channel definition contains `fromuser=sake`, `password=arigato`, and `host=thathostoverthere.tld`.

An extension number is often attached after the address information, like this:

```
exten => s,1,Dial(IAX2/user:pass@otherend.com/500)
```

This asks the far end to connect the call to extension 500 in the context in which the channel arrived. The extension is not required by `Dial( )`, as the information in the remote

* The fact that Asterisk will happily connect IAX, SIP, H.323, Skinny, PRI, FX(O/S), and anything else is amazing, but possibly the most amazing of all is the Local channel. By allowing a single `Dial( )` command to connect to multiple Local channels, one `Dial( )` event can trigger a multitude of completely independent and unique actions in other parts of the dialplan. The power of this concept is truly revolutionary and has to be experienced to be believed.

end's channel configuration file may be used, or the remote server will pass the call to the `s` extension in the context in which the call came in. Ultimately, the far end controls what happens to the call—you can only request a specific treatment.

If no *`ring-timeout`* is specified, the channel will ring indefinitely. This is not always a bad thing, so don't feel you need to set it—just be aware that "indefinitely" could mean a very long time. *`ring-timeout`* is specified in seconds. The ring timeout always follows the addressing information, like this:

```
exten => s,1,Dial(IAX2/user:pass@otherend.com/500,ring-timeout)
```

Much of the power of the `Dial( )` application is in the flags. These are assigned following the addressing and timeout information, like this:

```
exten => s,1,Dial(IAX2/user:pass@otherend.com/500,60,flags)
```

Here's something important to note: if you don't have a timeout specified, and you want to assign flags, you must still assign a spot for the timeout. You do this by adding an extra comma in the spot where the timeout would normally go, like this:

```
exten => s,1,Dial(IAX2/user:pass@otherend.com/500,,flags)
```

The valid flags that may be used with the `Dial( )` application are:

`d` Allows the user to dial a one-digit extension while waiting for a call to be answered. The call will then exit to that extension (either in the current context, if it exists, or in the context specified by `${EXITCONTEXT}`).

`t` Permits the called party to transfer a call by pressing the # key. Please note that if this option is used, reinvites are disabled, as Asterisk needs to monitor the call to detect when the called party presses the # key.

`T` Permits the caller to transfer a connected call by pressing the # key. Again, note that if this option is used, reinvites are disabled, as Asterisk needs to monitor the call to detect when the caller presses the # key.

`w` Permits the called user to start and stop recording the call audio to disk by pressing the `automon` sequence (as configured in *features.conf*). If the variable `TOUCH_MONITOR` is set, its value will be passed as the arguments to the `Monitor( )` application when recording is started. If it is not set, the default values of `WAV||m` are passed to `Monitor( )`.

`W` Permits the calling user to record the call audio to disk by pressing the `automon` sequence (as configured in *features.conf*).

`f` Forces the Caller ID to be set as the extension of the line making or redirecting the outgoing call. This is done because some PSTN providers will not allow the Caller ID to be set to anything other than that which is assigned to you. For example, if you had a PRI, you would use the `f` flag to override any Caller ID set locally on a SIP phone.

`o` Uses the Caller ID received on the inbound leg of the call for the Caller ID on the outbound leg of the call. This is useful if you are accepting a call and then forwarding it to another destination, but you wish to pass the Caller ID from the inbound leg of the call instead of overwriting it with the local Caller ID settings. This is the default behavior on Asterisk versions prior to 1.2.

r Indicates ringing to the calling party, without passing any audio until the call is answered. This flag is not normally required to indicate ringing, as Asterisk will signal ringing if a channel is actually being called.

m[*class*]
Provides music to the calling party until the call is answered. You may also optionally indicate the Music on Hold class.

M(*x*[^*arg*])
Executes the macro *x* upon the connection of a call, optionally passing arguments delimited by ^. The macro can also set the MACRO_RESULT channel variable to one of the following:

ABORT
Hangs up both legs of the call

CONGESTION
Acts as if the line encountered congestion

BUSY
Acts as if the line was busy (goes to n+101, where n is the current priority)

CONTINUE
Hangs up the called party and continues on in the dialplan

GOTO:<context>^<extension>^<priority>
Transfers the call to the specified destination

h Allows the called user to hang up the channel by pressing *.

H Allows the calling user to hang up the channel by pressing *.

C Resets the Call Detail Record for the call. Since the CDR time is set to when you Answer( ) the call, you may wish to reset the CDR so the end user is not billed for the time prior to the Dial( ) application being invoked.

P[(*x*)]
Sets the privacy mode, optionally specifying *x* as the family/key value in the local AstDB. Useful for accepting calls based on a blacklist (explicitly denying calls from listed numbers) or whitelist (explicitly accepting calls from listed numbers). See also LookupBlacklist( ).

g Goes on in the context if the destination channel hangs up.

G(*context^extension^priority*)
Transfers both parties to the specified destination, if the call is answered.

A(*x*)
Plays an announcement to the called party; *x* is the filename of the sound file to play as the announcement.

D([*called*][:*calling*])
Sends DTMF digits after the call has been answered, but before the call is bridged. The *called* parameter is passed to the called party, and the *calling* parameter is passed to the calling party. Either parameter may be used individually.

L(*x*[:*y*][:*z*])
Limits the call to *x* milliseconds, warning when *y* milliseconds are left and repeating every *z* milliseconds until the limit is reached. The *x* parameter is required; the *y* and *z*

parameters are optional. The following special variables may also be set to provide additional control:

`LIMIT_PLAYAUDIO_CALLER=yes|no`
: Specifies whether to play sounds to the caller

`LIMIT_PLAYAUDIO_CALLEE=yes|no`
: Specifies whether to play sounds to the callee

`LIMIT_TIMEOUT_FILE=`*`filename`*
: Specifies which file to play when time is up

`LIMIT_CONNECT_FILE=`*`filename`*
: Specifies which file to play when call begins

`LIMIT_WARNING_FILE=`*`filename`*
: Specifies the file to play if the argument *y* is defined

`n`
: Prevents jumping to priority `n+101` (where `n` is the number of the current priority) if all channels are deemed busy.

A call may also be parked instead of being transferred (which is done with the `t` or `T` flags). Calls are normally parked by transferring them to extension 700, but that's configurable in the *features.conf* file.

The `Dial( )` application sets the following variables upon exiting:

`DIALEDTIME`
: The total time elapsed from execution of `Dial( )` until completion.

`ANSWEREDTIME`
: The total time elapsed during the call.

`DIALSTATUS`
: The status of the call, set as one of the following values:

  `CHANUNAVAIL`
  : The channel is unavailable.

  `CONGESTION`
  : The channel returned a congestion signal, usually indicating that it was unable to complete the connection.

  `NOANSWER`
  : The channel did not answer in the time indicated by the ring-timeout option.

  `BUSY`
  : The dialed channel is currently busy.

  `ANSWER`
  : The channel answered the call.

  `CANCEL`
  : The call was cancelled.

```
; dial a seven-digit number on Zap channel 4
exten => 123,1,Dial(Zap/4/2317154)

; dial the same number, but this time only have it ring for 10 seconds
; before continuing on with the dialplan
```

```
exten => 124,1,Dial(Zap/4/2317154,10)
exten => 124,2,Playback(im-sorry)
exten => 124,3,Hangup( )

; dial the same number, but this time with no timeout, and using the
; t, T, and m flags
exten => 125,1,Dial(Zap/4/2317154,,tTm)

; dial extension 500 at a remote host (over the IAX protocol), using
; the specified username and password
exten => 126,1,Dial(IAX/username:password@remotehost/500)

; dial a number, but limit the call to 5 minutes (300,000 milliseconds)
; start warning the caller 4 minutes (240,000 milliseconds) into the call,
; and repeat the warning every 30 seconds (30,000 milliseconds)
exten => 127,1,Dial(Zap/4/2317154,,L[300000:240000:30000])
```

## DigitTimeout( )

Sets the maximum timeout between digits

`DigitTimeout(`*seconds*`)`

Sets the maximum amount of time permitted between digit presses when the caller is entering an extension. If the time period specified by *seconds* elapses after the caller enters a digit, the extension will be considered complete and will be interpreted.

Note that if a valid extension is typed in it will not have to time out to be tested, so typically at the expiration of this timeout, the extension will be considered invalid (and thus control will be passed to the `i` extension, or, if it doesn't exist, the call will be terminated).

Always returns `0`.

```
exten => 123,1,DigitTimeout(3)
exten => 123,2,Background(enter-ext-of-person)
exten => i,1,Playaback(im-sorry)
exten => i,2,Goto(123,1)
```

### See Also

`AbsoluteTimeout( )`

## Directory( )

Provides a dialable directory of extensions

`Directory(`*vm-context*`[,`*dial-context*`[,`*options*`]])`

Presents users with a directory of extensions from which they may select by name. The list of names and extensions is discovered from *voicemail.conf*. The *vm-context* argument is required; it specifies the context of *voicemail.conf* to use.

The *dial-context* argument is the context to use for dialing the users, and it defaults to *vm-context* if unspecified. Currently, the only option that can be specified in the *options* argument is `f`, which causes the directory to match based on the first name in *voicemail.conf* instead of the last name.

If the user enters `0` (zero) and there exists an extension `o` (the lowercase letter o) in the current context, the call control will go to that extension. Entering * will exit similarly, but to the `a` extension, much like `Voicemail( )`'s behavior.

Returns `0` unless the user hangs up.

```
exten => *,1,Directory(default,incoming)
exten => #,1,Directory(default,incoming,f)
```

### See Also

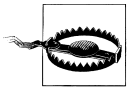
*voicemail.conf*

## DISA( )

Direct Inward System Access: allows inbound callers to make outbound calls

```
DISA(password[,context[,callerid[,mailbox[@vmcontext]]]])
DISA(password-file[,callerid[,mailbox[@vmcontext]]])
```

Allows outside callers to obtain an "internal" system dial tone and to place calls from it as if they were placing calls from within the switch. The user is given a dial tone, after which she should enter her passcode, followed by the pound sign (#). If the passcode is correct, the user is then given a system dial tone on which a call may be placed.



Obviously, this type of access has serious security implications, and *extreme* care must be taken not to compromise the security of your phone system.

The *password* argument is a numeric passcode that the user must enter to be able to make outbound calls. Using this syntax, all callers to this extension will use the same password. To allow users to use `DISA( )` without a password, put the string "no-password" instead of the password.

The *context* argument specifies the context in which the user will be dialing. If no context is specified, the `DISA( )` application defaults the context to `disa`.

The *callerid* argument specifies a new Caller ID string that will be used on the outbound call.

The *mailbox* argument is the mailbox number (and optional voicemail context, *vmcontext*) of a voicemail box. The caller will hear a stuttered dial tone if there are any new messages in the specified voicemail box.

Additionally, you may use an alternate syntax and pass the name of a global password file instead of the *password* and *context* arguments. On each line, the file may contain either a passcode, or a passcode and context, separated by a pipe character (|). If a context is not specified, the application defaults to the context named `disa`.

If the user login is successful, the application parses the dialed number in the specified *context*.

```
; allow outside callers to call 1-800 numbers, as long
; as they know the passcode. Set their Caller IDs to make
; it appear that they are dialing from within the company
[incoming]
```

```
exten => 123,1,DISA(4569,disa,"Company ABC" <(234) 123-4567>)

[disa]
exten => _18OONXXXXXX,1,Dial(Zap/4/${EXTEN})
```

## DumpChan( )

Dumps information about the calling channel to the console

```
DumpChan([min_verbose_level])
```

Displays information about the calling channel, as well as a listing of all channel variables. If *min_verbose_level* is specified, output is displayed only when the verbosity level is currently set to that number or greater.

Always returns 0.

```
exten => s,1,Answer( )
exten => s,2,DumpChan( )
exten => s,3,Background(enter-ext-of-person)
```

## DUNDiLookup( )

Looks up a phone number using DUNDi

```
DUNDiLookup(number[,context[,options]])
```

Looks up the given phone *number* in the *context* specified, or in the reserved e164 context if not specified. On completion, the variables ${DUNDTECH} and ${DUNDDEST} will contain the appropriate technology and destination to access the number. If no answer was found, and the priority n+101 (where n is the current priority) exists, execution will continue at that priority.

The *options* argument is currently ignored.

Returns -1 if the channel is hung up during the lookup, or 0 otherwise.

```
; look up a number via DUNDi, and dial it
exten => 123,1,DUNDiLookup(8885551212)
exten => 123,2,Dial(${DUNDITECH}/${DUNDDEST})
; if DUNDi lookup fails, dial it on a Zap channel instead
exten => 123,102,Dial(Zap/4/1888551212)
```

### See Also

ENUMLookup( )

## EAGI( )

See AGI( ).

## Echo( )

Echoes inbound audio back to the caller

Echo( )

Echoes audio read from the channel back to the channel. This application is often used to test the latency and voice quality of a VoIP link. The caller may press the # key to exit.

Returns 0 if the user exits with the # key, or -1 if the user hangs up.

```
exten => 123,1,Echo( )
exten => 123,2,Playback(vm-goodbye)
```

### See Also

Milliwatt( )

## EndWhile( )

Ends a while loop

EndWhile( )

Returns to the previously called While( ) application. See While( ) for a complete description of how to use a while loop.

```
exten => 123,1,Set(COUNT=1)
exten => 123,2,While($[ ${COUNT} < 5 ])
exten => 123,3,SayNumber(${COUNT})
exten => 123,4,EndWhile( )
```

### See Also

While( ), GotoIf( )

## ENUMLookup( )

Looks up a phone number in ENUM

ENUMLookup(*number*)

Looks up the telephone number specified by *number* via ENUM, and sets the variable ENUM with the result. For VoIP URIs, this variable will look like TECHNOLOGY/URI.

A good SIP, H.323, IAX, or IAX2 entry will result in normal-priority handling, whereas a good TEL entry will increase the priority by 51 (if the priority exists). If the lookup was *not* successful and there exists a priority n+101 (where n is the current priority), that priority will be taken next.

Currently, the only recognized ENUM services are SIP, H.323, IAX, IAX2, and TEL.

Returns -1 on hangup or 0 on completion, regardless of whether the lookup was successful.

```
; look up the phone number
exten => 123,1,ENUMLookup(8885551212)
; go to priority 2 on VoIP record
exten => 123,2,Dial(${ENUM})
; otherwise, go to priority 52 on TEL record
exten => 123,52,Dial(Zap4/${ENUM})
; otherwise, go to priority 102 because the lookup failed
exten => 123,102,Playback(im-sorry)
```

### See Also

DUNDiLookup( )

## Eval( )

Evaluates any Asterisk variables located within a string

Eval(*newvar=string*)

Processes the given *string* and evaluates any variables contained in the string. The resulting value is assigned to the variable *newvar*.

This application is used in situations where a string is used in the dialplan, but any variables contained within it need to be evaluated first. This is often the case when the string is retrieved from a database or other external source.

```
; go through some convoluted steps to create a string that contains
; the unparsed variable ${UNIQUEID}
exten => 123,1,Set(ONE=\$)
exten => 123,2,Set(TWO=$[{UNIQUEID}])
; print the values to the console, to make sure it hasn't been parsed
exten => 123,3,NoOp(${ONE}${TWO})
; now evaluate the variables in the string
exten => 123,4,Eval(TEST=${ONE}${TWO})
; print the result to the console
exten => 123,5,NoOp(${TEST})
```

### See Also

Exec( ), ExecIf( )

## Exec( )

Executes an Asterisk application dynamically

Exec(*appname*(*arguments*))

Allows an arbitrary application to be invoked even when not hard-coded into the dialplan. Returns whatever value the Asterisk *application* returns, or -2 when the called application cannot be found. The *arguments* are passed to the called *application*.

This application allows you to dynamically call applications by pulling them from a database or other external source.

```
exten => 123,1,Set(MYAPP=SayDigits(12345))
exten => 123,2,Exec(${MYAPP})
```

### See Also

Eval( ), ExecIf( )

## ExecIf( )

Conditionally executes an Asterisk application

`ExecIf(`*`expression`*`,`*`application`*`,`*`arguments`*`)`

If *`expression`* is true, executes the given *`application`* with *`arguments`* as its arguments, and returns the result. For more information on standard Asterisk expressions, see Chapter 6 or the *README.variables* file in the *doc/* subdirectory of the Asterisk source.

If *`expression`* is false, execution continues at the next priority.

```
exten => 123,1,ExecIf($[ ${CALLERIDNUM} = 101],SayDigits,12345)
exten => 123,2,SayDigits(6789)
```

### See Also

`Exec( )`, `Eval( )`

## FastAGI( )

Executes an AGI-compliant script across a network connection

`FastAGI(agi://`*`hostname`*`[:`*`port`*`][/`*`script`*`],`*`args`*`)`

Executes an AGI-compliant program across the network. This application is very similar to `AGI( )`, except that it calls a specially written FastAGI script across a network connection. The main purposes for using FastAGI are to offload CPU-intensive AGI scripts to remote servers and to help reduce AGI script startup times (the FastAGI program is already running before Asterisk connects to it).

`FastAGI( )` tries to connect directly to the running FastAGI program, which must already be listening for connections on the specified *`port`* on the server specified by *`hostname`*. If *`port`* is not specified, it defaults to port 4573. If *`script`* is specified, it is passed to the FastAGI program as the `agi_network_script` variable. The arguments specified by *`args`* will be passed to the program.

See *agi/fastagi-test* in the Asterisk source directory for a sample FastAGI script. This should serve as a good roadmap for writing your own FastAGI programs.

Returns `-1` if the application requested a hangup, or `0` on a non-hangup exit.

```
; connect to the sample fastagi-test program, which must already be running
; on the local machine
exten => 123,1,Answer( )
exten => 123,2,FastAGI(agi://localhost)

; connect to a FastAGI script on a host named "calvin" on port 8000, and pass along
; a script name of "testing", with the argument "12345"
exten => 124,1,Answer( )
exten => 124,2,FastAGI(agi://calvin:8000/testing,12345)
```

### See Also

`AGI( )`, `DeadAGI( )`

## Festival( )

Uses the Festival text-to-speech engine to read text to the caller

`Festival(`*`text`*`[,`*`intkeys`*`])`

Connects to the locally running Festival server, sends it the text specified by *`text`*, and plays the resulting sound file back to the user. This application allows the caller to press a key (specified by *`intkeys`*) to immediately stop the playback and return the value of *`intkeys`*. If *`intkeys`* is set to `any`, `Festival( )` will send control of the channel to the extension entered by the user.

See Chapter 10 for more in-depth information on using Festival with Asterisk.

You must start the Festival server before starting Asterisk, and you must use the `Answer( )` application to answer the channel before calling `Festival( )`.

For more information on using Festival from within Asterisk, see the *README.festival* file located in the *contrib/* subdirectory of the Asterisk source.

```
exten => 123,1,Answer( )
exten => 123,2,Festival('This is sample speech from Festival',#)
```

## Flash( )

Flashes a Zap trunk

`Flash( )`

Sends a flash on a Zap channel. This is only a hack for people who want to perform transfers and other actions that require a flash via an AGI script. It is generally quite useless otherwise.

Returns `0` on success or `-1` if this is not a Zap trunk.

```
exten => 123,1,Flash( )
```

## ForkCDR( )

Creates an additional CDR from the current call

`ForkCDR( )`

Creates an additional Call Detail Record for the remainder of the current call.

This application is often used in calling-card applications to distinguish the inbound call (the original CDR) from the billable call time (the second CDR).

```
exten => 123,1,Answer( )
exten => 123,2,ForkCDR( )
exten => 123,3,Playback(tt-monkeys)
exten => 123,4,Hangup( )
```

### See Also

`AppendCDRUserField( )`, `NoCDR( )`, `ResetCDR( )`, `SetCDRUserField( )`

## GetCPEID( )

Gets the CPE ID from an ADSI-capable telephone

```
GetCPEID( )
```

Obtains the CPE ID and other information and displays it on the Asterisk console. This information is often needed in order to properly set up *zapata.conf* for on-hook operations with ADSI-capable telephones.

Returns `-1` on hangup only.

```
; use this extension to get the necessary information to set up ADSI
; telephones
exten => 123,1,GetCPEID( )
```

### See Also

`ADSIProg( )`, *adsi.conf*, *zapata.conf*

## GetGroupCount( )

Counts the number of members in a particular group

```
GetGroupCount([group][@category])
```

Counts the members in the given *group* (and optional *category*) and sets the `${GROUPCOUNT}` variable to the corresponding value. If no *group* is specified, the current channel's group is used.

Use `SetGroup( )` to assign a call as a member of a particular group.

Always returns `0`.

```
; say the number of callers in the tech-support group
exten => 123,1,GetGroupCount(tech-support)
exten => 123,2,SayNumber(${GROUPCOUNT})
```

### See Also

`CheckGroup( )`, `GetGroupMatchCount( )`, `SetGroup( )`

## GetGroupMatchCount( )

Counts the number of members in all groups matching the specified pattern

```
GetGroupMatchCount(groupmatch[@category])
```

Counts the number of members in all groups matching the regular expression specified by *groupmatch*. The result is stored in the `${GROUPCOUNT}` variable.

Note that standard regular expressions are used in the *groupmatch* argument.

Always returns `0`.

```
; get the count of members in any group that starts with tech
exten => 123,1,GetGroupMatchCount(tech.*)
exten => 123,2,SayNumber($GROUPMATCH)
```

### See Also

`CheckGroup( )`, `GetGroupCount( )`, `SetGroup( )`

## Goto( )

Sends the call to the specified priority, extension, and context

```
Goto([[context,]extension,]priority)
Goto(named_priority)
```

Sends control of the current channel to the specified *priority*, optionally setting the destination *extension* and *context*.

Optionally, you can use the application to go to the named priority specified by the *named_priority* argument. Named priorities only work within the current extension.

Always returns 0, even if the given context, extension, or priority is invalid.

```
exten => 123,1,Answer( )
exten => 123,2,Set(COUNT=1)
exten => 123,3,SayNumber(${COUNT})
exten => 123,4,Set(COUNT=$[ ${COUNT} + 1 ])
exten => 123,5,Goto(3)

; same as above, but using a named priority
exten => 124,1,Answer( )
exten => 124,2,Set(COUNT=1)
exten => 124,3(repeat),SayNumber(${COUNT})
exten => 124,4,Set(COUNT=$[ ${COUNT} + 1 ])
exten => 124,5,Goto(repeat)
```

### See Also

GotoIf( ), GotoIfTime( )

## GotoIf( )

Conditionally goes to the specified priority

```
GotoIf(condition?label1:label2)
```

Sends the call to *label1* if *condition* is true or to *label2* if *condition* is false. Either *label1* or *label2* may be omitted (in that case, we just don't take the particular branch), but not both.

A label can be any one of the following:

- A priority, such as 10
- An extension and a priority, such as 123,10
- A context, extension, and priority, such as incoming,123,10
- A named priority within the same extension, such as passed

Each type of label is explained in the example below.

```
[globals]
; set TEST to something else besides 101 to see what GotoIf( )
; does when the condition is false
TEST=101
;
[incoming]
; set a variable
; go to priority 10 if ${TEST} is 101, otherwise go to priority 20
```

```
exten => 123,1,GotoIf($[ ${TEST} = 101 ]?10:20)
exten => 123,10,Playback(the-monkeys-twice)
exten => 123,20,Playback(tt-somethingwrong)
;
; same thing as above, but this time we'll specify an extension
; and a priority for each label
exten => 124,1,GotoIf($[ ${TEST} = 101 ]?123,10:123,20)
;
; same thing as above, but these labels have a context, extension, and
; priority
exten => 125,1,GotoIf($[ ${TEST} = 101 ]?incoming,123,10:incoming,123,20)
;
; same thing as above, but this time we'll go to named priorities
exten => 126,1,GotoIf($[ ${TEST} = 101 ]?passed:failed)
exten => 126,15(passed),Playback(the-monkeys-twice)
exten => 126,25(failed),Playback(the-monkeys-twice)
```

### See Also

`Goto( )`, `GotoIfTime( )`

## GotoIfTime( )

Conditionally branches, depending on the time and day

`GotoIfTime(`*`times`*`,`*`days_of_week`*`,`*`days_of_month`*`,`*`months`*`?`*`label`*`)`

Branches to the specified extension, if the current time matches the specified time. Each of the elements may be specified either as * (for always) or as a range.

The arguments to this application are:

*`times`*
: Time ranges, in 24-hour format

*`days_of_week`*
: Days of the week (`mon`, `tue`, `wed`, `thu`, `fri`, `sat`, `sun`)

*`days_of_month`*
: Days of the month (`1`–`31`)

*`months`*
: Months (`jan`, `feb`, `mar`, `apr`, etc.)

```
; If we're open, then go to the open context
; We're open from 9am to 6pm Monday through Friday
exten => s,1,GotoIfTime(09:00-17:59,mon-fri,*,*?open,s,1)
; We're also open from 9am to noon on Saturday
exten => s,2,GotoIfTime(09:00-11:59,sat,*,*?open,s,1)
; Otherwise, we're closed
exten => s,3,Goto(closed,s,1)
```

### See Also

`GotoIf( )`

## Hangup( )

Unconditionally hangs up the current channel

`Hangup( )`

Unconditionally hangs up the current channel.

Always returns `-1`.

```
exten => 123,1,Answer( )
exten => 123,2,Playback(im-sorry)
exten => 123,3,Hangup( )
```

### See Also

`Answer( )`

## HasNewVoicemail( )

Conditionally branches if there is new voicemail in the indicated voicemail box

`HasNewVoicemail(`*`vmbox`*`[@`*`context`*`][:`*`folder`*`][,`*`varname`*`])`

Similar to `HasVoicemail( )`. This application branches to priority `n+101` (where `n` is the current priority) if there is new (unheard) voicemail in the voicemail box indicated by *`vmbox`*. The *`context`* argument corresponds to the voicemail context, and *`folder`* corresponds to a voicemail folder. If the voicemail folder is not specified, it defaults to the *INBOX* folder. If the *`varname`* argument is present, `HasNewVoicemail( )` assigns the number of messages in the specified folder to that variable.

```
; check to see if there's unheard voicemail in INBOX of mailbox 123
; in the default voicemail context
exten => 123,1,Answer( )
exten => 123,2,HasNewVoicemail(123@default,COUNT)
exten => 123,3,Playback(vm-youhave)
exten => 123,4,Playback(vm-no)
exten => 123,5,Playback(vm-messages)
exten => 123,103,Playback(vm-youhave)
exten => 123,104,SayNumber($COUNT)
exten => 123,105,Playback(vm-messages)
```

### See Also

`HasVoicemail( )`, `MailboxExists( )`

## HasVoicemail( )

Conditionally branches if there is voicemail in the indicated voicemail box

`HasVoicemail(`*`vmbox`*`[@`*`context`*`][:`*`folder`*`][|`*`varname`*`])`

Branches to priority `n+101` (where `n` is the current priority) if there is voicemail in the voicemail box indicated by *`vmbox`*. The *`context`* argument corresponds to the voicemail context, and *`folder`* corresponds to a voicemail folder. If the folder is not specified, it defaults to the *INBOX* folder. If the *`varname`* argument is passed, this application assigns the number of messages in the specified folder to that variable.

```
; check to see if there's any voicemail at all in INBOX of mailbox 123
; in the default voicemail context
```

```
exten => 123,1,Answer( )
exten => 123,2,HasVoicemail(123@default,COUNT)
exten => 123,3,Playback(vm-youhave)
exten => 123,4,Playback(vm-no)
exten => 123,5,Playback(vm-messages)
exten => 123,103,Playback(vm-youhave)
exten => 123,104,SayNumber($COUNT)
exten => 123,105,Playback(vm-messages)
```

### See Also

HasNewVoicemail( ), MailboxExists( )

## IAX2Provision( )

Provisions a calling IAXy device

IAX2Provision([*template*])

Provisions a calling IAXy device (assuming that the calling entity is an IAXy) with the given *template*. If no template is specified, the default template is used. IAXy provisioning templates are defined in the *iaxprov.conf* configuration file.

Returns -1 on error or 0 on success.

```
; provision IAXy devices with the default template when they dial this extension
exten => 123,1,IAX2Provision(default)
```

## ImportVar( )

Sets a variable based on a channel variable from a different channel

ImportVar(*newvar=channel*,*variable*)

Sets variable *newvar* to *variable* as evaluated on the specified *channel* (instead of the current channel). If *newvar* is prefixed with _, single inheritance is assumed. If prefixed with __, infinite inheritance is assumed.

```
; read the Caller ID information from channel Zap/1
exten => 123,1,Answer( )
exten => 123,1,ImportVar(cidinfo=Zap/1,CALLERID)
```

### See Also

Set( )

## LookupBlacklist( )

Performs a lookup of a Caller ID name/number from the blacklist database

LookupBlacklist( )

Looks up the Caller ID number on the active channel in the Asterisk database (family blacklist). If the number is found, and if there exists a priority n+101 (where n is the priority of the current instance), the channel will be set up to continue at that priority level. Otherwise, the application returns 0. If no Caller ID was received on the channel, it does nothing.

To add to the blacklist from the Asterisk CLI, type **database put blacklist *name/number***.

```
; send blacklisted numbers to an endless loop
; otherwise, dial the number defined by the variable ${JOHN}
exten => s,1,Answer( )
exten => s,2,LookupBlacklist( )
exten => s,3,Dial(${JOHN})
exten => s,103,Playback(tt-allbusy)
exten => s,104,Wait(10)
exten => s,105,Goto(103)
```

## LookupCIDName( )

Performs a lookup of a Caller ID name from the AstDB

`LookupCIDName( )`

Uses the Caller ID number on the active channel to retrieve the Caller ID name from the AstDB (family `cidname`). This application does nothing if no Caller"*ID was received on the channel. This is useful if you do not subscribe to Caller ID name delivery, or if you want to change the Caller ID names on some incoming calls.

Always returns `0`.

```
; look up the Caller ID information from the AstDB, and pass it along
; to Jane's phone
exten => 123,1,Answer( )
exten => 123,2,LookupCIDName( )
exten => 123,3,Dial(SIP/Jane)
```

## Macro( )

Calls a previously defined macro

`Macro(`*macroname*`,`*arg1*`,`*arg2*`...)`

Executes a macro defined in the context named `macro-`*macroname*, jumping to the `s` extension of that context and executing each step, then returning when the steps end.

The calling extension, context, and priority are stored in `${MACRO_EXTEN}`, `${MACRO_CONTEXT}`, and `${MACRO_PRIORITY}`, respectively. Arguments *arg1*, *arg2*, etc. become `${ARG1}`, `${ARG2}`, etc. in the macro context.

`Macro( )` returns `-1` if any step in the macro returns `-1`, and `0` otherwise. If `${MACRO_OFFSET}` is set at termination, this application will attempt to continue at priority `MACRO_OFFSET+n+1` if such a step exists, and at `n+1` otherwise. (In both cases, `n` stands for the current priority.)

If you call the `Goto( )` application inside of the macro, the macro will terminate and control will go to the destination of the `Goto( )`.

```
; define a macro to count down from the specified value
[macro-countdown]
exten => s,1,Set(COUNT=${ARG1})
exten => s,2,While($[ ${COUNT} > 0])
exten => s,3,SayNumber(${COUNT})
exten => s,4,Set(COUNT=$[ ${COUNT} - 1 ])
exten => s,5,EndWhile( )
```

```
; call our macro with two different values
[example]
exten => 123,1,Macro(countdown,10)
exten => 124,1,Macro(countdown,5)
```

### See Also

Goto( ), Chapter 6

## MailboxExists( )

Conditionally branches if the specified voicemail box exists

MailboxExists(*mailbox*[@*context*])

Conditionally branches to priority n+101 (where n is the current priority) if the voicemail box specified by the *mailbox* argument exists. You may pass a voicemail *context* if the mailbox is not in the default voicemail context.

```
exten => 123,1,Answer( )
exten => 123,2,MailboxExists(123@default)
exten => 123,3,Playback(im-sorry)
exten => 123,103,Voicemail(u123)
```

### See Also

HasVoicemail( ), HasNewVoicemail( )

## Math( )

Performs mathematical operations and returns the result

Math(*returnvar,number1 operator number2*)

Performs a floating-point calculation on *number1* to *number2*, and assigns the result to the variable named *returnvar*. Valid operators are +, -, /, *, %, <, >, >=, <=, and ==, and they behave as their C equivalents.

Always returns 0.

```
; add two numbers, and say the result
exten => 123,1,Math(SUM,2+2)
exten => 123,2,SayNumber(${SUM})

; subtract two numbers, and say the difference
exten => 124,1,Math(DIFFERENCE,5-3)
exten => 124,2,SayNumber(${DIFFERENCE})
```

## MeetMe( )

Puts the caller into a MeetMe conference bridge

MeetMe([*confno*][,[*options*][,*pin*])

Joins the caller on the current channel into the MeetMe conference specified by the *confno* argument. If the conference number is omitted, the user will be prompted to enter one.

The *options* string may contain zero or more of the characters in the following list.

m Sets monitor-only mode (listen only, no talking).

t Sets talk-only mode (talk only, no listening).

T Sets talker detection (sent to Manager interface and MeetMe list).

i Announces user join/leave.

p Allows user to exit the conference by pressing #.

X Allows user to exit the conference by entering a valid single-digit extension (set via the variable ${MEETME_EXIT_CONTEXT}), or the number of an extension in the current context if that variable is not defined.

d Dynamically adds conference.

D Dynamically adds conference, prompting for a PIN.

e Selects an empty conference.

E Selects an empty pinless conference.

v Sets video mode.

r Records conference (as ${MEETME_RECORDINGFILE} using format ${MEETME_RECORDINGFORMAT}). The default filename is *meetme-conf-rec*-${CONFNO}-${UNIQUEID} and the default format is *.wav*.

q Sets quiet mode (don't play enter/leave sounds).

M Enables Music on Hold when the conference has a single caller.

x Closes the conference when the last marked user exits.

w Waits until the marked user enters the conference.

b Runs the AGI script specified in ${MEETME_AGI_BACKGROUND}. Default: *conf-background.agi*. (Note: this does not work with non-Zap channels in the same conference.)

s Presents the menu (user or admin) when * is received ("send" to menu).

a Sets admin mode.

A Sets marked mode.

If the *pin* argument is passed, the caller must enter that pin number to successfully enter the conference.

MeetMe( ) returns 0 if the caller presses # to exit (see option p); otherwise, it returns -1.

A suitable Zaptel timing interface must be installed for MeetMe conferencing to work.

```
exten => 123,1,Answer( )
; add the caller to conference number 501 with pin 1234
exten => 123,2,MeetMe(501,DpM,1234)
```

## See Also

MeetMeAdmin( ), MeetMeCount( )

## MeetMeAdmin( )

Performs MeetMe conference administration

MeetMeAdmin(*confno*,*command*[,*pin*])

Runs the specified MeetMe administration *command* on the specified conference. The *command* may be one of the following (note that the *pin* argument is used only for the k option):

K Kicks all users out of the conference

k Kicks one user (with the specified PIN as the third argument) out of the conference

e Ejects the last user that joined

L Locks the conference

l Unlocks the conference

M Mutes the conference

m Unmutes the conference

N Mutes the entire conference (except admin)

n Unmutes the entire conference (except admin)

```
; mute conference 501
exten => 123,1,MeetMeAdmin(501,M)

; kick user with PIN number 1234 from conference 501
exten => 124,1,MeetMeAdmin(501,k,1234)
```

### See Also

MeetMe( ), MeetMeCount( )

## MeetMeCount( )

Counts the number of participants in a MeetMe conference

MeetMeCount(*confno*[,*variable*])

Plays back the number of users in the MeetMe conference identified by *confno*. If a variable is specified by the *variable* argument, playback will be skipped and the count will be assigned to *variable*.

Returns 0 on success or -1 on a hangup.

```
; count the number of users in conference 501, and assign that number to ${COUNT}
exten => 123,1,MeetMeCount(501,COUNT)
```

### See Also

MeetMe( ), MeetMeAdmin( )

## Milliwatt( )

Generates a 1,000-Hz tone

```
Milliwatt( )
```

Generates a constant 1,000-Hz tone at 0 dbm (mu-law). This application is often used for testing the audio properties of a particular channel.

```
; generate a milliwatt tone for testing
exten => 123,1,Milliwatt( )
```

### See Also

`Echo( )`

## Monitor( )

Monitors (records) the audio on the current channel

`Monitor([`*file_format*`[:`*urlbase*`][,`*fname_base*`[,`*options*`]])`

Starts monitoring a channel. The channel's input and output voice packets are logged to files until the channel hangs up or monitoring is stopped by the `StopMonitor( )` application.

`Monitor( )` takes the following arguments:

*file_format*
: Specifies the file format. If not set, defaults to `wav`.

*fname_base*
: If set, changes the filename used to the one specified.

*options*
: One of two options can be specified:

`m`
: When the recording ends, mix the two leg files into one and delete the original leg files. If the variable `${MONITOR_EXEC}` is set, the application referenced in it will be executed instead of *soxmix*, and the raw leg files will *not* be deleted automatically. *soxmix* (or `${MONITOR_EXEC}`) is handed three arguments: the two leg files and the filename for the target mixed file, which is the same as the leg filenames but without the in/out designator. If `${MONITOR_EXEC_ARGS}` is set, the contents will be passed on as additional arguments to `${MONITOR_EXEC}`. Both `${MONITOR_EXEC}` and the `m` flag can be set from the administrator interface

`b`
: Don't begin recording unless a call is bridged to another channel.

Returns `-1` if monitor files can't be opened or if the channel is already monitored; otherwise, returns `0`.

```
exten => 123,1,Answer( )
; record the current channel, and mix the audio channels at the end of
; recording
exten => 123,2,Monitor(wav,monitor_test,mb)
exten => 123,3,SayDigits(12345678901234567890)
exten => 123,4,StopMonitor( )
```

### See Also

`ChangeMonitor( )`, `StopMonitor( )`

## MP3Player( )

Plays an MP3 file or stream

`MP3Player(`*`location`*`)`

Uses the *mpg123* program to play the given *`location`* to the caller. The specified *`location`* can be either a filename or a valid URL. The caller can exit by pressing any key.

The correct version of *mpg123* must be installed for this application to work properly. Asterisk currently works best with *mpg123-0.59r*.

Returns `-1` on hangup; otherwise, returns `0`.

```
exten => 123,1,Answer( )
exten => 123,2,MP3Player(test.mp3)

exten => 123,1,Answer( )
exten => 123,2,MP3Player(http://server.tld/test.mp3)
```

## MusicOnHold( )

Plays Music on Hold indefinitely

`MusicOnHold(`*`class`*`)`

Plays hold music specified by *`class`*, as configured in *musiconhold.conf*. If omitted, the default music class for the channel will be used. You can use the `SetMusicOnHold( )` application to set the default music class for the channel.

Returns `-1` on hangup; otherwise, does not return.

```
; transfer telemarketers to this extension to keep them busy
exten => 123,1,Answer( )
exten => 123,2,Playback(tt-allbusy)
exten => 123,3,MusicOnHold(default)
```

### See Also

`SetMusicOnHold( )`, `WaitMusicOnHold( )`

## NBScat( )

Plays an NBS local stream

`NBScat( )`

Uses the `nbscat8k` program to listen to the local Network Broadcast Sound (NBS) stream. (For more information, see the *nbs* module in Digium's CVS server.) The caller can exit by pressing any key.

Returns -1 on hangup; otherwise, does not return.

```
exten => 123,1,Answer( )
exten => 123,2,NBScat( )
```

## NoCDR( )

Disables Call Detail Records for the current call

NoCDR( )

Disables CDRs for the current call.

```
; don't log calls to 555-1212
exten => 5551212,1,Answer( )
exten => 5551212,2,NoCDR( )
exten => 5551212,3,Dial(Zap/4/5551212)
```

### See Also

AppendCDRUserField( ), ForkCDR( ), SetCDRUserField

## NoOp( )

Does nothing

NoOp(*text*)

Does nothing—this application is simply a placeholder. As a side effect, the application evaluates *text* and prints the result to the Asterisk command-line interface, which can be useful for debugging.

You don't have to place quotes around the text. If quotes are placed within the brackets, they will show up on the console.

```
exten => 123,1,NoOp(CallerID is ${CALLERID})
```

## Park( )

Parks the current call

Park(*exten*)

Parks the current call (typically in combination with a supervised transfer to determine the parking space number). This application is always registered internally and does not need to be explicitly added into the dialplan, although you should include the parkedcalls context. Parking configuration is set in *features.conf*.

```
; park the caller in parking space 701
include => parkedcalls
exten => 123,1,Answer( )
exten => 123,2,Park(701)
```

### See Also

ParkAndAnnounce, ParkedCall( )

## ParkAndAnnounce( ) — Parks the current call and announces the call over the specified channel

ParkAndAnnounce(*template*,*timeout*,*channel*[,*return_context*])

Parks the current call in the parking lot and announces the call over the specified *channel*. The *template* is a colon-separated list of files to announce; the word PARKED is replaced with the parking space number of the call. The *timeout* argument is the time in seconds before the call returns to the *return_context*. The *channel* argument is the channel to call to make the announcement. Console/dsp calls the console. The *return_context* argument is a GoTo( )-style label to jump the call back into after timeout, which defaults to n+1 (where n is the current priority) in the *return_context* context.

```
include => parkedcalls
exten => 123,1,Answer( )
exten => 123,2,ParkAndAnnounce(vm-youhave:a:pbx-transfer:at:vm-extension:
PARKED,120,Console/dsp)
exten => 123,3,Playback(vm-nobodyavail)
exten => 123,4,Playback(vm-goodbye)
exten => 123,5,Hangup( )
```

### See Also

Park( ), ParkedCall( )

## ParkedCall( ) — Answers a parked call

ParkedCall(*exten*)

Connects the caller to the parked call in the parking space identified by *exten*. This application is always registered internally and does not need to be explicitly added into the dialplan, although you should include the parkedcalls context.

```
; pick up the call parked in parking space 701
exten => 123,1,Answer( )
exten => 123,2,ParkedCall(701)
```

### See Also

Park( ), ParkAndAnnounce( )

## PauseQueueMember( ) — Temporarily blocks a queue member from receiving calls

PauseQueueMember([*queuename*],*interface*)

Pauses (blocks calls for) a queue member. The specified interface will be paused in the given queue. This prevents any calls from being sent from the queue to the interface until it is unpaused by the UnpauseQueueMember( ) application or the Manager interface. If no *queuename* is given, the interface is paused in every queue it is a member of. If the *interface* is not in the named queue, or if no queue is given and the *interface* is not in any queue, it will jump to priority n+101 (where n is the current priority), if it exists.

Returns -1 if the interface is not found and no extension to jump to exists; otherwise, returns 0.

```
exten => 123,1,PauseQueueMember(,SIP/300)
exten => 124,1,UnpauseQueueMember(,SIP/300)
```

### See Also

UnpauseQueueMember( )

## Playback( )

Plays the specified audio file to the caller

Playback(*filename*[,*options*])

Plays back a given filename to the caller. The filename should not contain the file extension, as Asterisk will automatically choose the audio file with the lowest conversion cost. Zero or more *options* may also be included. The skip option causes the playback of the message to be skipped if the channel is not in the "up" state (i.e., it hasn't yet been answered). If skip is specified, the application will return immediately should the channel not be off-hook. Otherwise, unless noanswer is specified, the channel will be answered before the sound file is played. (Not all channels support playing messages while still on-hook.) Returns -1 if the channel was hung up. If the file does not exist, jumps to priority n+101 (where n is the current priority), if it exists.

```
exten => 123,1,Answer( )
exten => 123,2,Playback(tt-weasels)
```

### See Also

Background( )

## Playtones( )

Plays a tone list

Playtones(*tonelist*)

Plays a tone list. Execution immediately continues with the next step, while the tones continue to play. The *tonelist* is either the tone name defined in the *indications.conf* configuration file, or a specified list of frequencies and durations. See *indications.conf* for a description of the specification of a tone list.

Use the StopPlaytones( ) application to stop the tones playing.

```
; play a busy signal for two seconds, and then a congestion tone for two seconds
exten => 123,1,Playtones(busy)
exten => 123,2,Wait(2)
exten => 123,3,StopPlaytones( )
exten => 123,4,Playtones(congestion)
exten => 123,5,Wait(2)
exten => 123,6,StopPlaytones( )
exten => 123,7,Goto(1)
```

### See Also

StopPlaytones( ), *indications.conf*, Busy( ), Congestion( ), Progress( ), Ringing( )

## Prefix( ) — Prepends the specified digits to the current extension and goes to the resulting extension

```
Prefix(digits)
```

Prefixes the current extension with the digit string specified by *digits* and continues processing at the next priority for the new extension. So, for example, if priority 1 of extension 1212 is `Prefix(555)`, 555 will be prepended to 1212 and the next step executed will be priority 2 of extension 5551212. If you switch into an extension that has no priority `n+1` (where `n` is the current priority), Asterisk will treat it as though the user dialed an invalid extension.

Always returns `0`.

```
exten => 1212,1,Prefix(555)
exten => 5551212,2,SayDigits(${EXTEN})
```

### See Also

`Suffix( )`

## PrivacyManager( ) — Requires a caller to enter his or her phone number, if no Caller ID information is received

```
PrivacyManager( )
```

If no Caller ID is received, answers the channel and asks the caller to enter his or her phone number. By default, the caller is given three attempts. If after three attempts the caller has not entered at least a 10-digit phone number, and if there exists a priority `n+101` (where `n` is the current priority), the channel will be set up to continue at that priority level. Otherwise, it returns `0`. If Caller ID was received on the channel, `PrivacyManager( )` does nothing..

The *privacy.conf* configuration file changes the functionality of the `PrivacyManger( )` application. It contains the following two lines:

`maxretries`
: Specifies the maximum number of attempts the caller is allowed to input a Caller ID number (default: `3`)

`minlength`
: Specifies the minimum allowable digits in the input Caller ID number (default: `10`)

```
exten => 123,1,Answer( )
exten => 123,2,PrivacyManager( )
exten => 123,3,Dial(Zap/1)
exten => 123,103,Playback(im-sorry)
exten => 123,104,Playback(vm-goodbye)
```

### See Also

`Zapateller( )`

## Progress( )

Indicates progress

`Progress( )`

Requests that the channel indicate that in-band progress is available to the user.

Always returns `0`.

```
; indicate progress to the calling channel
exten => 123,1,Progress( )
```

### See Also

`Busy( )`, `Congestion( )`, `Ringing( )`, `Playtones( )`

## Queue( )

Places the current call into the specified call queue

`Queue(`*`queuename`*`[,`*`options`*`[,`*`URL`*`[,`*`announceoverride`*`[,`*`timeout`*`]]]])`

Places an incoming call into the call queue specified by *queuename*, as defined in *queues.conf*.

The *options* argument may contain zero or more of the following characters:

`t` Allows the called user to transfer the call

`T` Allows the calling user to transfer the call

`d` Specifies a data-quality (modem) call (minimum delay)

`h` Allows callee to hang up by hitting *

`H` Allows caller to hang up by hitting *

`n` Disallows retries on the timeout; exits this application and goes to the next step

`r` Rings instead of playing MoH

In addition to being transferred, a call may be parked and then picked up by another user.

The *announceoverride* argument overrides the standard announcement played to queue agents before they answer the specified call.

The optional *URL* will be sent to the called party if the channel supports it.

The *timeout* will cause the queue to fail out after a specified number of seconds, checked between each *queues.conf* *timeout* and *retry* cycle.

Returns `-1` if the originating channel hangs up, or if the call is bridged and either of the parties in the bridge terminates the call. If the queue is full, does not exist, or has no members, returns `0`.

```
; place the caller in the techsupport queue
exten => 123,1,Answer( )
exten => 123,2,Queue(techsupport,t)
```

## Random( )

Conditionally branches, based upon a probability

Random([*probability*]:[[*context*,]*extension*,]*priority*)

Conditionally jumps to the specified *priority* (and optional *extension* and *context*), based on the specified *probability*. *probability* should be an integer between 1 and 100. The application will jump to the specified destination *priority* percent of the time.

```
; test your luck over and over again
exten => 123,1,Random(20:lucky,1)
exten => 123,2,Goto(unlucky,1)

exten => lucky,1,Playback(good)
exten => lucky,2,Goto(123,1)

exten => unlucky,1,Playback(bad)
exten => unlucky,2,Goto(123,1)
```

## Read( )

Reads DTMF digits from the caller and assign the result to a variable

Read(*variable*[,*filename*][,*maxdigits*][,*option*][,*attempts*][,*timeout*])

Reads a #-terminated string of digits a certain number of times from the user into the given *variable*.

Other arguments include:

*filename*
: Specifies the file to play before reading digits.

*maxdigits*
: Sets the maximum acceptable number of digits. If this argument is specified, the application stops reading after *maxdigits* have been entered (without requiring the user to press the # key). Defaults to 0– (no limit, wait for the user to press the # key). Any value below 0 means the same. The maximum accepted value is 255.

*option*
: Specify skip to return immediately if the line is not answered, or *noanswer* to read digits even if the line is not answered.

*attempts*
: If greater than 1, that many attempts will be made in the event that no data is entered.

*timeout*
: If greater than 0, that value will override the default timeout.

Returns -1 on hangup or error and 0 otherwise.

```
; read a two-digit number and repeat it back to the caller
exten => 123,1,Read(NUMBER,,2)
exten => 123,2,SayNumber(${NUMBER})
exten => 123,3,Goto(1)
```

### See Also

SendDTMF( )

## RealTime

Looks up information from the RealTime configuration handler

RealTime(*family*,*colmatch*,*value*[,*prefix*])

Uses the RealTime configuration handler system to read data into channel variables. All unique column names (from the specified *family*) will be set as channel variables, with an optional *prefix* to the name (e.g., a prefix of var_ would make the column name become the variable ${var_name}).

```
; retrieve all columns from the sipfriends table where the name column
; matches "John", and prefix all the variables with "John_"
exten => 123,1,RealTime(sipfriends,name,John,John_)
; now, let's read the value of the column named "port"
exten => 123,2,SayNumber(${John_port})
```

### See Also

RealTimeUpdate( )

## RealTimeUpdate( )

Updates a value via the RealTime configuration handler

RealTimeUpdate(*family*,*colmatch*,*value*,*newcol*,*newval*)

Uses the RealTime configuration handler system to update a value. The column *newcol* in *family* matching column *colmatch=value* will be updated to *newval*.

```
; this will update the port column in the sipfriends table to a new
; value of 5061, where the name column matches "John"
exten => 123,1,RealTimeUpdate(sipfriends,name,John,port,5061)
```

### See Also

RealTime( )

## Record( )

Records channel audio to a file

Record(*filename*:*format*,*silence*[,*maxduration*][,*options*]) (in Asterisk 1.0.x)
Record(*filename*.*format*,*silence*[,*maxduration*][,*options*]) (in Asterisk 1.2.x)

Records audio from the channel into the given *filename*. If the file already exists, it will be overwritten.

Optional arguments include:

*format*
: Specifies the format of the file type to be recorded. Valid formats include: g723, g729, gsm, h263, ulaw, alaw, vox, wav, and WAV.

*silence*
: Specifies the number of seconds of silence to allow before returning.

*maxduration*
: Specifies the maximum recording duration, in seconds. If missing or 0, there is no maximum.

*options*
: May contain any of the following letters:

s
: Skip recording if the line is not yet answered.

n
: Do not answer, but record anyway if the line is not yet answered.

a
: Append the recording to the existing recording rather than replacing it.

t
: Use the alternate * terminator key instead of the default #.

If the filename contains %d, these characters will be replaced with a number incremented by one each time the file is recorded.

The user can press # to terminate the recording and continue to the next priority.

Returns -1 when the user hangs up.

```
; record the caller's name
exten => 123,1,Playback(pls-rcrd-name-at-tone)
exten => 123,2,Record(/tmp/name:gsm,3,30)
exten => 123,3,Playback(/tmp/name)
```

## RemoveQueueMember( )

Dynamically removes queue members

RemoveQueueMember(*queuename*[,*interface*])

Dynamically removes the specified *interface* from the *queuename* call queue. If *interface* is not specified, this application removes the current interface from the queue.

If the interface is not in the queue and there exists a priority n+101 (where n is the current priority), the application will jump to that priority. Otherwise, it will return an error.

Returns -1 if there is an error.

```
; remove SIP/3000 from the techsupport queue
exten => 123,1,RemoveQueueMember(techsupport,SIP/3000)
```

### See Also

AddQueueMember( )

## ResetCDR( )

Resets the Call Detail Record

ResetCDR([*options*])

Causes the Call Detail Record to be reset. If the w option is specified, a copy of the current CDR will be stored before the current CDR is zeroed out.

Always returns 0.

```
; write a copy of the current CDR record, and then reset the CDR
exten => 123,1,Answer( )
exten => 123,2,Playback(tt-monkeys)
```

```
exten => 123,3,ResetCDR(w)
exten => 123,4,Playback(tt-monkeys)
```

### See Also

ForkCDR( ), NoCDR( )

## ResponseTimeout( )

Sets maximum timeout for awaiting response from caller

ResponseTimeout(*seconds*)

Sets the maximum amount of time permitted after falling through a series of priorities for a channel in which the caller may begin typing an extension. If the caller does not type an extension in this amount of time, control will pass to the t extension, if it exists; if not, the call will be terminated.

Always returns 0.

```
; allow callers three seconds to make a choice, before sending them
; to the 't' extension
exten => s,1,Answer( )
exten => s,2,ResponseTimeout(3)
exten => s,3,Background(enter-ext-of-person)

exten => t,1,Playback(im-sorry)
exten => t,1,Playback(goodbye)
```

### See Also

AbsoluteTimeout( ), DigitTimeout( )

## RetryDial( )

Attempts to place a call, and retries on failure

RetryDial(*announce*,*sleep*,*loops*,*technology*/*resource*[&*Technology2*/*resource2*...][,*timeout*][,*options*][,*URL*])

Attempts to place a call. If no channel can be reached, plays the file defined by *announce*, waiting *sleep* seconds to retry the call. If the specified number of attempts matches *loops*, the call will continue with the next priority in the dialplan. If *loops* is set to 0, the call will retry endlessly.

While waiting, a one-digit extension may be dialed. If that extension exists in either the context defined in ${EXITCONTEXT} (if defined) or the current one, the call will transfer to that extension immediately.

All arguments after *loops* are passed directly to the Dial( ) application.

```
; attempt to dial the number three times via IAX, retrying every five seconds
exten => 123,1,RetryDial(priv-trying,5,3,IAX2/VOIP/8885551212,30)
; if the caller presses 9 while waiting, dial the number on the Zap/4 channel
exten => 9,1,RetryDial(priv-trying,5,3,Zap/4/8885551212,30)
```

### SeeAlso

Dial( )

## Ringing( )

Indicates ringing tone

```
Ringing( )
```

Requests that the channel indicate ringing tone to the user. It is up to the channel driver to specify exactly how ringing is indicated.

Note that this application does not actually provide audio ringing to the caller. Use the `Playtones( )` application to do this.

Always returns `0`.

```
; indicate that the phone is ringing, even though it isn't
exten => 123,1,Ringing( )
exten => 123,2,Wait(5)
exten => 123,3,Playback(tt-somethingwrong)
```

### See Also

`Busy( )`, `Congestion( )`, `Progress( )`, `Ringing( )`, `Playtones( )`

## SayAlpha( )

Spells a string

```
SayAlpha(string)
```

Spells out the passed *string*, using the current language setting for the channel. See the `SetLanguage( )` application to change the current language.

```
exten => 123,1,SayAlpha(ABC123XYZ)
```

### See Also

`SayDigits( )`, `SayNumber( )`, `SayPhonetic( )`, `SetLanguage( )`

## SayDigits( )

Says the passed digits

```
SayDigits(digits)
```

Says the passed digits, using the current language setting for the channel. See the `SetLanguage( )` application to change the current language.

```
exten => 123,1,SayDigits(1234)
```

### See Also

`SayAlpha( )`, `SayNumber( )`, `SayPhonetic( )`, `SetLanguage( )`

## SayNumber( )

Says the specified number

```
SayNumber(digits[,gender])
```

Says the specified number, using the current language setting for the channel. See the `SetLanguage( )` application to change the current language.

Currently, syntax for the following languages is supported:

`da` Danish
`de` German
`en` English
`es` Spanish
`fr` French
`it` Italian
`nl` Dutch
`no` Norwegian
`pl` Polish
`pt` Portuguese
`se` Swedish
`tw` Taiwanese

If the current language supports different genders, you can pass the *gender* argument to change the gender of the spoken number. You can use the following *gender* arguments:

- Use the *gender* arguments `f` for female, `m` for male, and `n` for neuter in European languages such as Portuguese, French, Spanish, and German.
- Use the *gender* argument `c` for commune and `n` for neuter in Nordic languages such as Danish, Swedish, and Norwegian.
- Use the *gender* argument `p` for plural enumerations in German.

For this application to work in languages other than English, you must have the appropriate sounds for the language you wish to use.

```
; say the number in English
exten => 123,1,SetLanguage(en)
exten => 123,2,SayNumber(1234)
```

### See Also

`SayAlpha( )`, `SayDigits( )`, `SayPhonetic( )`, `SetLanguage( )`

## SayPhonetic( )

Spells the specified string phonetically

`SayPhonetic(`*`string`*`)`

Spells the specified *string* using the NATO phonetic alphabet.

```
exten => 123,1,SayPhonetic(asterisk)
```

### See Also

`SayAlpha( )`, `SayDigits( )`, `SayNumber( )`

## SayUnixTime( )

Says the specified time in a custom format

SayUnixTime([*unixtime*][,[*timezone*][,*format*]])

Speaks the specified time according to the specified time zone and format. The arguments are:

*unixtime*
: The time, in seconds, since January 1, 1970. May be negative. Defaults to now.

*timezone*
: The time zone. See */usr/share/zoneinfo/* for a list. Defaults to the machine default.

*format*
: The format in which the time is to be spoken. See *voicemail.conf* for a list of formats. Defaults to "ABdY 'digits/at' IMp".

Returns 0, or -1 on hangup.

```
exten => 123,1,SayUnixTime(,,IMp)
```

## SendDTMF( )

Sends arbitrary DTMF digits to the channel

SendDTMF(*digits*[,*timeout_ms*])

Sends the specified DTMF digits on a channel. Valid DTMF digits include 0-9, *, #, and A-D. You may also use the letter w as a digit, which indicates a 500-millisecond wait. The *timeout_ms* argument is the amount of time between digits, in milliseconds. If not specified, *timeout_ms* defaults to 250 milliseconds.

Returns 0 on success or -1 on a hangup.

```
exten => 123,1,SendDTMF(3212333w222w366w3212333322321,250)
```

### See Also

Read( )

## SendImage( )

Sends an image file

SendImage(*filename*)

Sends an image on a channel, if image transport is supported. If the channel does not support image transport, and there exists a priority n+101 (where n is the current priority), execution will continue at that step. Otherwise, execution will continue at the next priority level.

Returns 0 if the image was sent correctly or if the channel does not support image transport; otherwise, returns -1.

```
exten => 123,1,SendImage(logo.jpg)
```

### See Also

SendText( ), SendURL( )

## SendText( )

Sends text to the channel

SendText(*text*)

Sends *text* on a channel, if text transport is supported. If the channel does not support text transport, and there exists a priority n+101 (where n is the current priority), execution will continue at that step. Otherwise, execution will continue at the next priority level.

Returns 0 if the text was sent correctly or if the channel does not support text transport; otherwise, returns -1.

```
exten => 123,1,SendText(Welcome to Asterisk)
```

### See Also

SendImage( ), SendURL( )

## SendURL( )

Sets a variable to the specified value

SendURL(*URL*[,*option*])

Requests that the client go to the specified URL. If the client does not support HTML transport, and there exists a priority n+101 (where n is the number of the current priority), execution will continue at that step. Otherwise, execution will continue at the next priority level.

Returns 0 if the URL was sent correctly or if the channel does not support HTML transport; otherwise, returns -1.

If the option wait is specified, execution will wait for an acknowledgment that the URL has been loaded before continuing and will return -1 if the peer is unable to load the URL.

```
exten => 123,1,SendURL(www.asterisk.org,wait)
```

### See Also

SendImage( ), SendText( )

## Set( )

Sends

Set(*n=value*)

Sets the variable *n* to the specified *value*. If the variable name is prefixed with _, single inheritance is assumed. If the variable name is prefixed with __, infinite inheritance is assumed. Inheritance is used when you want the outgoing channel to inherit the variable from the dialplan.

Variables set with this application are valid only in the current channel. Use the SetGlobalVar( ) application to set global variables.

```
; set a variable called DIALTIME, then use it
exten => 123,1,SetVar(DIALTIME=20)
exten => 123,1,Dial(Zap/4/5551212,,${DIALTIME})
```

### See Also

SetGlobalVar( ), *README.variables*

## SetAccount( )

Sets the account code in the Call Detail Record

SetAccount(*account*)

Sets the account code in the Call Detail Record, for billing purposes.

Always returns 0.

```
; set the account code to 4321 before dialing the boss
exten => 123,1,SetAccount(4321)
exten => 123,2,Dial(${BOSS})
```

### See Also

SetAMAFlags( ), SetCDRUserField( ), AppendCDRUserField( )

## SetAMAFlags( )

Sets AMA flags in the Call Detail Record

SetAMAFlags(*flag*)

Sets the AMA flags in the Call Detail Record for billing purposes, overriding any AMA settings in the channel configuration files. Valid choices are default, omit, billing, and documentation.

Always returns 0.

```
exten => 123,1,SetAMAFlags(billing)
```

### See Also

SetAccount( ), SetCDRUserField( ), AppendCDRUserField( )

## SetCallerID( )

Sets the Caller ID for the channel

SetCallerID(*clid*[,*a*])

Sets the Caller ID on the channel to a specified value. If the a argument is passed, ANI is also set to the specified value.

Always returns 0.

```
; set both the Caller ID and ANI
exten => 123,1,SetCallerID("John Q. Public <8885551212>",a)
```

### See Also

SetCIDName( ), SetCIDNum( )

## SetCallerPres( )

Sets Caller ID presentation flags

SetCallerPres(*presentation*)

Sets the Caller ID presentation flags on a Q931 PRI connection.

Valid presentations are:

`allowed_not_screened`
: Presentation Allowed, Not Screened

`allowed_passed_screen`
: Presentation Allowed, Passed Screen

`allowed_failed_screen`
: Presentation Allowed, Failed Screen

`allowed`
: Presentation Allowed, Network Number

`prohib_not_screened`
: Presentation Prohibited, Not Screened

`prohib_passed_screen`
: Presentation Prohibited, Passed Screen

`prohib_failed_screen`
: Presentation Prohibited, Failed Screen

`prohib`
: Presentation Prohibited, Network Number

`unavailable`
: Number Unavailable

Always returns `0`.

```
exten => 123,1,SetCallerPres(allowed_not_screened)
exten => 123,2,Dial(Zap/g1/8885551212)
```

### SeeAlso

SetCallerID( )

## SetCDRUserField( )

Sets the Call Detail Record user field

SetCDRUserField(*value*)

Sets the CDR user field to the specified *value*. The CDR user field is an extra field that you can use for data not stored anywhere else in the record. CDR records can be used for billing purposes or for storing other arbitrary data about a particular call.

```
exten => 123,1,SetCDRUserField(testing)
exten => 123,2,Playback(tt-monkeys)
```

### See Also

AppendCDRUserField( ), SetAccount( ), SetAMAFlags( )

## SetCIDName( )

Sets the Caller ID name on the channel

SetCIDName(*cname*[,a])

Sets the Caller ID name on the current channel to *cname*, while preserving the original Caller ID number. This is useful for providing additional information to the called party. If the a option is used, ANI is also set.

Always returns 0.

```
exten => 123,1,SetCIDName("John Q. Public")
```

### See Also

SetCallerID( ), SetCIDNum( )

## SetCIDNum( )

Sets the Caller ID number for a channel

SetCIDNum(*cnum*[,a])

Sets the Caller ID number on the current channel to the number specified by *cnum*, while preserving the original Caller ID name. This is useful for providing additional information to the called party. The application sets ANI as well if the a flag is used.

Always returns 0.

```
exten => 123,1,SetCIDNum(8885551212)
```

### See Also

SetCIDName( ), SetCallerID( )

## SetGlobalVar( )

Sets a global variable to the specified value

SetGlobalVar(*n*=*value*)

Sets a global variable called *n* to the specified *value*. Global variables are available across channels.

```
; set the NUMRINGS global variable to 3
exten => 123,1,SetGlobalVar(NUMRINGS=3)
```

### See Also

SetVar( )

## SetGroup( )

Sets the channel group to the specified value

SetGroup(*groupname*[@*category*])

Sets the channel group to the specified *groupname* value. Equivalent to Set(GROUP=*group*). Used in conjunction with CheckGroup( ) to limit the number of calls accessing a particular resource. A group *category* may also be set.

Always returns 0.

```
; limit the number of concurrent receptionist calls to three
exten => s,1,SetGroup(receptionist)
exten => s,2,2,CheckGroup(3)
exten => s,3,Dial(${RECEPTIONIST})
exten => s,103,VoiceMail(u${RECEPTION_VM})
```

### See Also

CheckGroup( ), GetGroupCount( ), GetGroupMatchCount( )

## SetLanguage( )

Sets the channel's language

SetLanguage(*language*)

Sets the channel language to *language*. This information is used for the syntax in generation of numbers, and to choose a natural language file when available. For example, if *language* is set to fr and the file *demo-congrats* is requested to be played, the file *fr/demo-congrats* will be played if it exists; if not, the normal *demo-congrats* file will be played.

Always returns 0.

```
exten => s,1,SetLanguage(fr)
exten => s,2,SayNumber(1234)
exten => s,3,Playback(enter-ext-of-person)
```

## SetMusicOnHold( )

Sets the default Music on Hold class for the current channel

SetMusicOnHold(*class*)

Sets the default *class* for Music on Hold for the current channel. When Music on Hold is activated, this class will be used to select which music is played. Classes are defined in the configuration file *musiconhold.conf*.

```
exten=s,1,Answer( )
exten=s,2,SetMusicOnHold(default)
exten=s,3,WaitMusicOnHold( )
```

### See Also

WaitMusicOnHold( ), *musiconhold.conf*, MusicOnHold( )

## SetRDNIS( )

Sets the RDNIS number on the current channel

SetRDNIS(*cnum*)

Sets the Redirected Dial Number ID Service (RDNIS) number on a call to the value specified by *cnum*. RDNIS is supported only on certain PRI lines.

Always returns 0.

```
exten => 123,1,SetRDNIS(8885551212)
exten => 123,2,Dial(Zap/4/5551234)
```

## SetVar( )

Sets a variable to the specified value

SetVar(*n*=*value*)

Sets the variable *n* to the specified *value*. If the variable name is prefixed with _, single inheritance is assumed. If the variable name is prefixed with __, infinite inheritance is assumed. Inheritance is used when you want the outgoing channel to inherit the variable from the dialplan. Reprecated in favor of Set(), which has the same syntax.

Variables set with this application are valid only in the current channel. Use the SetGlobalVar( ) application to set global variables.

```
; set a variable called DIALTIME, then use it
exten => 123,1,SetVar(DIALTIME=20)
exten => 123,1,Dial(Zap/4/5551212,,${DIALTIME})
```

### See Also

SetGlobalVar( ), *README.variables*

## SIPAddHeader( )

Adds a SIP header to the outbound call

SIPAddHeader(*Header*: *Content*)

Adds a header to a SIP call placed with the Dial( ) application. A nonstandard SIP header should begin with X-, such as X-Asterisk-Accountcode:. Use this application with care—adding the wrong headers may cause any number of problems.

Always returns 0.

```
exten => 123,1,SIPAddHeader(X-Asterisk-Testing: Just testing!)
exten => 123,2,Dial(SIP/123)
```

### See Also

SIPGetHeader( )

## SIPDtmfMode( )

Changes the DTMF method for a SIP call

SIPDtmfMode(*method*)

Changes the DTMF method for a SIP call. The *method* can be either inband, info, or rfc2833.

```
exten => 123,1,SIPDtmfMode(rfc2833)
exten => 123,2,Dial(SIP/123)
```

### See Also

Appendix A

## SIPGetHeader( )

Gets a SIP header from an incoming SIP call

SIPGetHeader(*var=headername*)

Sets a channel variable named *var* to the content of the *headername* SIP header. Skips to priority n+101 (where n is the current priority) if the specified header does not exist.

```
; get the "To" header and assign it to the variable called TESTING
exten => 123,1,SIPGetHeader(TESTING=To)
```

### See Also

SIPAddHeader( )

## SoftHangup( )

Performs a soft hangup of the requested channel

SoftHangup(*technology/resource,options*)

Hangs up the requested channel. Always returns 0. The *options* argument may contain the letter a, which causes all channels on the specified device to be hung up. Currently, the *options* argument may contain only one letter: a. Supplying the a argument causes all channels on the specified device to be hung up.

```
; hang up all calls using Zap/4 so we can use it
exten => 123,1,SoftHangup(Zap/4,a)
exten => 123,2,Wait(2)
exten => 123,3,Dial(Zap/4/5551212)
```

### See Also

Hangup( )

## StopMonitor( )

Stops monitoring a channel

StopMonitor( )

Stops monitoring (recording) a channel. This application has no effect if the channel is not currently being monitored.

```
exten => 123,1,Answer( )
exten => 123,2,Monitor(wav,monitor_test,mb)
exten => 123,3,SayDigits(12345678901234567890)
exten => 123,4,StopMonitor( )
```

### See Also

Monitor( )

## StopPlaytones( )

Stops playing a tone list

StopPlaytones( )

Stops playing the currently playing tone list.

```
exten => 123,1,Playtones(busy)
exten => 123,2,Wait(2)
exten => 123,3,StopPlaytones( )
exten => 123,4,Playtones(congestion)
exten => 123,5,Wait(2)
exten => 123,6,StopPlaytones( )
exten => 123,7,Goto(1)
```

### See Also

`Playtones( )`, *indications.conf*

## StripLSD( )

Strips the specified number of trailing (least significant) digits from the current extension

`StripLSD(`*count*`)`

Strips the trailing *count* digits from the channel's associated extension and continues processing at the next priority for the resulting extension. So, for example, if priority 1 of extension 5551212 is `StripLSD(4)`, the last 4 digits will be stripped from 5551212 and the next step executed will be priority 2 of extension 555. If you switch into an extension that has no priority `n+1` (where `n` is the current priority), the PBX will treat it as though the user dialed an invalid extension.

Always returns `0`.

This application is deprecated and has been replaced with the substring expression `${EXTEN:`*X*`:`*Y*`}`.

```
exten => 5551212,1,StripLSD(4)
exten => 555,2,SayDigits(${EXTEN})

; a better way of doing the same thing
exten => 5551234,1,SayDigits(${EXTEN::3})
```

### See Also

`StripMSD( )`, *README.variables*, variable substring syntax

## StripMSD( )

Strips the specified number of leading (most significant) digits from the current extension

`StripMSD(`*count*`)`

Strips the leading *count* digits from the channel's associated extension and continues processing at the next priority for the resulting extension. So, for example, if priority 1 of extension 5551212 is `StripMSD(3)`, the first 3 digits will be stripped from 5551212 and the next step executed will be priority 2 of extension 1212. If you switch into an extension that has no priority `n+1` (where `n` is the current priority), the PBX will treat it as though the user dialed an invalid extension.

Always returns `0`.

This application is deprecated and has been replaced with the substring expression ${EXTEN:*X*:*Y*}.

```
exten => 5551212,1,StripMSD(3)
exten => 1212,2,SayDigits(${EXTEN})

; a better way of doing the same thing
exten => 5551234,1,SayDigits(${EXTEN:3})
```

### See Also

StripLSD( ), *README.variables*, variable substring syntax

## SubString( )

Saves substring digits in a given variable

SubString(*variable*=*string_of_digits*,*count1*,*count2*)

Assigns the substring of *string_of_digits* to a given variable. The parameter *count1* may be positive or negative. If it's positive, we skip the first *count1* digits from the left. If it's negative, we move *count1* digits from the end of the string to the left. The parameter *count2* indicates how many digits to take from the point that *count1* placed us. If *count2* is negative, that many digits are omitted from the end.

This application is deprecated. Instead, use ${EXTEN:*X*:*Y*}.

```
; here are some examples using SubString( ):
;assign the area code (3 first digits) to variable TEST
exten => 8885551212,1,SubString(TEST=8885551212,0,3)
; assign the last 7 digits to variable TEST
exten => 8885551212,1,SubString(TEST=8885551212,-7,7)
; assign all but the last 4 digits to variable TEST
exten => 8885551212,1,SubString(TEST=8885551212,0,-4)
;
; and here are the preferred alternatives:
;assign the area code (3 first digits) to variable TEST
exten => 8885551212,1,Set(TEST=${EXTEN::3})
; assign the last 7 digits to variable TEST
exten => 8885551212,1,Set(TEST=${EXTEN:-7:7})
; assign all but the last 4 digits to variable TEST
exten => 8885551212,1,Set(TEST=${EXTEN:6}
```

## Suffix( )

Appends trailing digits to the current extension

Suffix(*digits*)

Appends the digit string specified by *digits* to the channel's associated extension and continues processing at the next priority for the new extension. So, for example, if priority 1 of extension 555 is Suffix(1212), 1212 will be appended to 555 and the next step executed will be priority 2 of extension 5551212. If you switch into an extension that has no priority n+1 (where n is the current priority), the PBX will treat it as though the user dialed an invalid extension.

Always returns 0.

```
exten => 555,1,Suffix(1212)
exten => 5551212,2,SayDigits(${EXTEN})
```

### See Also

Prefix( )

## System( )

Executes an operating system command

System(*command*)

Executes a *command* in the underlying operating system. If the command itself executes but is in error, and if there exists a priority n+101 (where n is the current priority), the execution of the dialplan will continue at that priority level.

This application is very similar to the TrySystem( ) application, except that it will return -1 if it is unable to execute the system command, whereas the TrySystem( ) application will always return 0.

```
exten => 123,1,System(echo hello > /tmp/hello.txt)
```

### See Also

TrySystem( )

## Transfer( )

Transfers the caller to a remote extension

Transfer(*exten*)

Requests that the remote caller be transferred to the given extension. If the transfer is *not* supported or successful and there exists a priority n+101 (where n is the current priority), that priority will be taken next.

```
; transfer calls from extension 123 to extension 130
exten => 123,1,Transfer(130)
```

## TrySystem( )

Tries to execute an operating system command

TrySystem(*command*)

Attempts to execute a *command* in the underlying operating system. If the command itself executes but is in error, and if there exists a priority n+101 (where n is the current priority), the execution of the dialplan will continue at that priority level.

This application is very similar to the System( ) application, except that it always returns 0, whereas the System( ) application will return -1 if it is unable to execute the system command.

```
exten => 123,1,TrySystem(echo hello > /tmp/hello.txt)
```

### See Also

System( )

## TXTCIDName( )

Looks up a caller's name from a DNS TXT record

TXTCIDName(*CallerID*)

Looks up a caller's name via DNS and sets the variable ${TXTCIDNAME}. TXTCIDNAME will either be blank or return the value found in the TXT record in DNS. This application looks up the number via the ENUM sources listed in *enum.conf*.

```
exten => 123,1,TXTCIDName(8662331454)
exten => 123,2,SayAlpha(${TXTCIDNAME})
exten => 123,3,Playback(vm-goodbye)
```

## UnpauseQueueMember( )

Unpauses a queue member

UnpauseQueueMember([*queuename*],*interface*)

Unpauses (resumes calls to) a queue member. This is the counterpart to PauseQueueMember( ), and it operates exactly the same way, except it unpauses instead of pausing the given interface.

```
exten => 123,1,PauseQueueMember(,SIP/300)
exten => 124,1,UnpauseQueueMember(,SIP/300)
```

### See Also

PauseQueueMember( )

## UserEvent( )

Sends an arbitrary event to the Manager interface

UserEvent(*eventname*[,*body*])

Sends an arbitrary event to the Manager interface, with an optional body representing additional arguments. The format of the event is:

```
Event: UserEvent<specified event name>
Channel: <channel name>
Uniqueid: <call uniqueid>
[body]
```

If the body is not specified, only the Event, Channel, and Uniqueid fields will be present.

Always returns 0.

```
exten => 123,1,UserEvent(BossCalled,${CALLERIDNAME} has called the boss!)
exten => 123,2,Dial(${BOSS})
```

### See Also

*manager.conf*, Asterisk Manager interface

## Verbose( )

Sends arbitrary text to verbose output

Verbose([*level*,]*message*)

Sends the specified *message* to verbose output. The *level* must be an integer value. If not specified, *level* defaults to 0.

Always returns 0.

```
exten => 123,1,Verbose(Somebody called extension 123)
exten => 123,2,Playback(extension)
exten => 123,3,SayDigits(${EXTEN})
```

## VMAuthenticate( )

Authenticates the caller from voicemail passwords

VMAuthenticate([*mailbox*][@*context*])

Behaves identically to the Authenticate( ) application, with the exception that the passwords are taken from *voicemail.conf*.

If *mailbox* is specified, only that mailbox's password will be considered valid. If *mailbox* is not specified, the channel variable ${AUTH_MAILBOX} will be set with the authenticated mailbox.

```
; authenticate off of any mailbox password, and tell us the matching
; mailbox number
exten => 123,1,VMAuthenticate( )
exten => 123,2,SayDigits(${AUTH_MAILBOX})
```

### See Also

Authenticate( ), *voicemail.conf*

## VoiceMail( )

Leaves a voicemail message in the specified mailbox

VoiceMail([s|u|b]*mailbox*[@*context*][&*mailbox*[@*context*]][...])

Leaves voicemail for a given *mailbox* (must be configured in *voicemail.conf*).

If the mailbox is preceded by s, instructions for leaving the message will be skipped. If it is preceded by u, the "unavailable" message (*/var/lib/asterisk/sounds/vm/exten/unavail*) will be played, if it exists. If the mailbox is preceded by b, the busy message will be played (that is, *busy* instead of *unavail*).

If the caller presses 0 (zero) during the prompt, the call jumps to the o (lower-case letter o) extension in the current context.

If the caller presses * during the prompt, the call jumps to extension a in the current context. This is often used to send the caller to a personal assistant.

If the requested mailbox does not exist, and there exists a priority n+101 (where n is the current priority), that priority will be taken next.

When multiple mailboxes are specified, the unavailable or busy message will be taken from the first mailbox specified.

Returns -1 on error or mailbox not found, or if the user hangs up; otherwise, returns 0.

```
; send caller to unavailable voicemail for mailbox 123
exten => 123,1,VoiceMail(u123)
```

### See Also

VoiceMailMain( ), *voicemail.conf*

## VoiceMailMain( )

Enters the voicemail system

VoiceMailMain([[s|p]*mailbox*][@*context*])

Enters the main voicemail system for the checking of voicemail. Passing the *mailbox* argument will stop the voicemail system from prompting the user for the mailbox number.

If the mailbox is preceded by the letter s, the password check will be skipped. If the mailbox is preceded by the letter p, the supplied mailbox will be prepended to the user's entry and the resulting string will be used as the mailbox number. This is useful for virtual hosting of voicemail boxes. If a *context* is specified, logins are considered in that voicemail context only.

Returns -1 if the user hangs up; otherwise, returns 0.

```
; go to voicemail menu for mailbox 123 in the default voicemail context
exten => 123,1,VoiceMailMain(123@default)
```

### See Also

VoiceMail( ), *voicemail.conf*

## Wait( )

Waits for a specified number of seconds

Wait(*seconds*)

Waits for the specified number of *seconds*, then returns 0. You can pass fractions of a second (e.g., 1.5 = 1.5 seconds).

```
; wait 1.5 seconds before playing the prompt
exten => s,1,Answer( )
exten => s,2,Wait(1.5)
exten => s,3,Background(enter-ext-of-person)
```

## WaitExten( )

Waits for an extension to be entered

WaitExten([*seconds*])

Waits for the user to enter a new extension for the specified number of seconds, then returns 0. You can pass fractions of a second (e.g., 1.5 = 1.5 seconds). If unspecified, the default extension timeout will be used.

```
; wait 15 seconds for the user to dial an extension
exten => s,1,Answer( )
exten => s,2,Playback(enter-ext-of-person)
exten => s,3,WaitExten(15)
```

## WaitForRing( )

Waits the specified number of seconds for a ring

`WaitForRing(`*`timeout`*`)`

Waits at least *`timeout`* seconds after the next ring has completed.

Returns `0` on success or `-1` on hangup.

```
; wait five seconds for a ring, and then send some DTMF digits
exten => 123,1,Answer( )
exten => 123,2,WaitForRing(5)
exten => 123,3,SendDTMF(1234)
```

## WaitForSilence( )

Waits for a specified amount of silence

`WaitForSilence(`*`wait`*`[,`*`repeat`*`])`

Waits for *`repeat`* instances of *`wait`* milliseconds of silence. If *`repeat`* is omitted, the application waits for a single instance of *`wait`* milliseconds of silence.

```
; wait for three instances of 300 ms of silence
exten => 123,WaitForSilence(300,3)
```

## WaitMusicOnHold( )

Waits the specified number of seconds, playing Music on Hold

`WaitMusicOnHold(`*`delay`*`)`

Plays hold music for the specified number of seconds. If no hold music is available the delay will still occur, but with no sound.

Returns `0` when done, or `-1` on hangup.

```
; allow caller to hear Music on Hold for five minutes
exten => 123,1,Answer( )
exten => 123,2,WaitMusicOnHold(300)
exten => 123,3,Hangup( )
```

### See Also

`SetMusicOnHold( )`, *musiconhold.conf*

## While( )

Starts a `while` loop

`While(`*`expr`*`)`

Starts a `while` loop. Execution will return to this point when `EndWhile( )` is called, until *`expr`* is no longer true. If a condition is met causing the loop to exit, it continues on past the `EndWhile( )`.

```
exten => 123,1,Set(COUNT=1)
exten => 123,2,While($[ ${COUNT} < 5 ])
exten => 123,3,SayNumber(${COUNT})
exten => 123,4,EndWhile( )
```

### See Also

`EndWhile( )`, `GotoIf( )`

## Zapateller( )

Uses a special information tone to block telemarketers

`Zapateller(`*`options`*`)`

Generates a special information tone to block telemarketers and other computer-dialed calls from bothering you.

The *`options`* argument is a pipe-delimited list of options. The following options are available:

`answer`
: Causes the line to be answered before playing the tone

`nocallerid`
: Causes Zapateller to play the tone only if no Caller ID information is available

```
; answer the line, and play the SIT tone if there is no Caller ID information
exten => 123,1,Zapateller(answer|nocallerid)
```

### See Also

`PrivacyManager( )`

## ZapBarge( )

Barges in on (monitors) a Zap channel

`ZapBarge([`*`channel`*`])`

Barges in on a specified Zap *`channel`*, or prompts if one is not specified. The people on the channel won't be able to hear you and will have no indication that their call is being monitored.

If *`channel`* is not specified, you will be prompted for the channel number. Enter **`4#`** for Zap/4, for example.

Returns `-1` when the caller hangs up and is independent of the state of the channel being monitored.

```
exten => 123,1,ZapBarge(Zap/2)
exten => 123,2,Hangup( )
```

### See Also

`ZapScan( )`

## ZapRAS( )

Executes the Zaptel ISDN Remote Access Server

`ZapRAS(`*`args`*`)`

Executes an ISDN RAS server using *pppd* on the current channel. The channel must be a clear channel (i.e., PRI source) and a Zaptel channel to be able to use this function. (No modem emulation is included.)

Your *pppd* must be patched to be Zaptel-aware. *args* is a pipe-delimited list of arguments.

Always returns `-1`.

This application is only for use on ISDN lines, and your kernel must be patched to support `ZapRAS( )`. You must also have *ppp* support in your kernel.

```
exten => 123,1,Answer( )
exten => 123,1,ZapRas(debug|64000|noauth|netmask|255.255.255.0|10.0.0.1:10.0.0.2)
```

## ZapScan( )

Scans Zap channels to monitor calls

`ZapScan([group])`

Allows a call center manager to monitor Zap channels in a convenient way. Use # to select the next channel, and use * to exit. You may limit scanning to a particular channel group by setting the *group* argument.

```
exten => 123,1,ZapScan( )
```

### See Also

`ZapBarge( )`