CHAPTER 6

More Dialplan Concepts

For a list of all the ways technology has failed to improve the quality of life, please press three.
—Alice Kahn

Alrighty. You've got the basics of dialplans down, and you're hoping there's more to come. Fear not; there is more—much more. If you don't have the last chapter sorted out yet, please go back and give it another read. We're building on what we've covered so far, and we need you to be comfortable with the material, as we're about to get into more advanced topics.

Expressions and Variable Manipulation

Before we dive further into dialplans, we need to introduce you to a few tricks that will greatly add to the power you can exercise with your dialplan. These constructs add incredible intelligence to your dialplan, by enabling it to make decisions based on all sorts of different criteria. Put on your thinking cap, and let's get started.

Basic Expressions

Expressions are combinations of variables, operators, and values that you put together to get a result. An expression can test values, alter strings, or perform mathematical calculations. Let's say we have a variable called `COUNT`. In plain English, two expressions using that variable might be "`COUNT` plus 1" and "`COUNT` divided by 2." Each of these expressions has a particular result or value, depending on the value of the given variable.

In Asterisk, expressions always begin with a dollar sign and an opening square bracket and end with a closing square bracket, as shown below:

```
$[expression]
```

Thus, we would write the above two examples like this:

```
$[${COUNT} + 1]
$[${COUNT} / 2]
```

When Asterisk encounters an expression in a dialplan, it replaces the entire expression with the resulting value. It is important to note that this takes place *after* variable substitution. To demonstrate, let's look at the following code:*

```
exten => 321,1,Set(COUNT=3)
exten => 321,2,Set(NEWCOUNT=$[${COUNT} + 1])
exten => 321,3,SayNumber(${NEWCOUNT})
```

In the first priority, we assign the value of `3` to the variable named `COUNT`.

In the second priority, only one application—`Set()`—is involved, but three things actually happen:

1. Asterisk substitutes `${COUNT}` with the number `3` in the expression. The expression effectively becomes this:

   ```
   exten => 321,2,Set(NEWCOUNT=$[3 + 1])
   ```

2. Next, Asterisk evaluates the expression, adding `1` to `3`, and replaces it with its computed value of `4`:

   ```
   exten => 321,2,Set(NEWCOUNT=4)
   ```

3. Finally, the value `4` is assigned to the `NEWCOUNT` variable by the `Set()` application.

The third priority simply invokes the `SayNumber()` application, which speaks the current value of the variable `${NEWCOUNT}` (set to the value `4` in priority two).

Try it out in your own dialplan.

Operators

When you create an Asterisk dialplan, you're really writing code in a specialized scripting language. This means that the Asterisk dialplan—like any programming language—recognizes symbols called *operators* that allow you to manipulate variables. Let's look at the types of operators that are available in Asterisk:

Boolean operators

These operators evaluate the "truth" of a statement. In computing terms, that essentially refers to whether the statement is something or nothing (non-zero or zero, true or false, on or off, and so on). The Boolean operators are:

* Remember that when you *reference* a variable, you can call it by its name, but when you refer to a variable's *value*, you have to use the dollar sign and brackets around the variable name.

`expr1 | expr2`
: This operator (called the “or” operator, or “pipe”) returns the evaluation of *`expr1`* if it is true (neither an empty string nor zero). Otherwise, it returns the evaluation of *`expr2`*.

`expr1 & expr2`
: This operator (called “and”) returns the evaluation of *`expr1`* if both expressions are true (i.e., neither expression evaluates to an empty string or zero). Otherwise, it returns zero.

`expr1 {=, >, >=, <, <=, !=} expr2`
: These operators return the results of an integer comparison if both arguments are integers; otherwise, they return the results of a string comparison. The result of each comparison is `1` if the specified relation is true, or `0` if the relation is false. (If you are doing string comparisons, they will be done in a manner that’s consistent with the current locale settings of your operating system.)

Mathematical operators

Want to perform a calculation? You’ll want one of these:

`expr1 {+, -} expr2`
: These operators return the results of the addition or subtraction of integer-valued arguments.

`expr1 {*, /, %} expr2`
: These return, respectively, the results of the multiplication, integer division, or remainder of integer-valued arguments.

Regular expression operator

You can also use the regular expression operator in Asterisk:

`expr1 : expr2`
: This operator matches *`expr1`* against *`expr2`*, where *`expr2`* must be a regular expression.* The regular expression is anchored to the beginning of the string with an implicit `^`.†

If the match succeeds and the pattern contains at least one regular expression subexpression—`\(...\)`—the string corresponding to `\1` is returned; otherwise, the matching operator returns the number of characters matched. If the match fails and the pattern contains a regular expression subexpression, the null string is returned; otherwise, `0` is returned.

* For more on regular expressions, grab a copy of the ultimate reference, Jeffrey Friedl’s *Mastering Regular Expressions* (O’Reilly; *http://www.oreilly.com/catalog/regex2/*) or visit *http://www.regular-expressions.info*.

† If you don’t know what a ^ has to do with regular expressions, you simply must obtain a copy of *Mastering Regular Expressions*. It will change your life!

The Asterisk parser is quite simple, so it requires that you put at least one space between the operator and any other values. Consequently, the following may not work as expected:

```
exten => 123,1,Set(TEST=$[2+1])
```

This would assign the variable `TEST` the string "2+1", instead of the value `3`. Instead, put spaces around the operator, like this:

```
exten => 234,1,Set(TEST=$[2 + 1])
```

To concatenate text onto the beginning or end of a variable, simply place them together in an expression, like this:

```
exten => 234,1,Set(NEWTEST=$[blah${TEST}])
```

Dialplan Functions

Dialplan functions are not a new concept. In Asterisk 1.2, they should be used where possible. Many applications that perform the same operation as a corresponding function will eventually be removed in favor of the function. Functions allow you to add more power to your expressions—you can think of them as being similar to operators, but more advanced. For example, dialplan functions allow you to calculate string lengths, dates and times, MD5 checksums, and so on, all from within a dialplan expression.

Syntax

Dialplan functions have the following basic syntax:

```
FUNCTION_NAME(argument)
```

Much like with variables, you reference a function's *name* as above, but you reference a function's *value* with the addition of a dollar sign, an opening curly brace, and a closing curly brace:

```
${FUNCTION_NAME(argument)}
```

Functions can also encapsulate other functions, like so:

```
${FUNCTION_NAME(${FUNCTION_NAME(argument)})}
 ^             ^ ^             ^        ^^^^
 1             2 3             4        4321
```

As you've probably already figured out, you must be very careful about making sure you have matching parentheses and braces. In the above example, we have labeled the opening parentheses and curly braces with numbers and their corresponding closing counterparts with the same numbers.

Examples of Dialplan Functions

Functions are often used in conjunction with the Set( ) application to either get or set the value of a variable. As a simple example, let's look at the LEN( ) function. This function calculates the string length of its argument. Let's calculate the string length of a variable and read back the length to the caller:

```
exten => 123,1,Set(TEST=example)
exten => 123,2,SayNumber(${LEN(${TEST})})
```

The above example would evaluate the string example as having seven characters, assign the number of characters to the variable length, and then speak the number to the user with the SayNumber( ) application.

Let's look at another simple example. If we wanted to set one of the various channel timeouts, we could use the TIMEOUT( ) function. The TIMEOUT( ) function accepts one of three arguments: absolute, digit, and response. Their corresponding applications are AbsoluteTimeout( ), DigitTimeout( ), and ResponseTimeout( ). To set the digit timeout with the TIMEOUT( ) function, we could use the Set( ) application, like so:

```
exten => s,1,Set(TIMEOUT(digit)=30)
```

Notice the lack of ${ } surrounding the function. Just as if we were assigning a value to a variable, we assign a value to a function without the use of the ${ } encapsulation.

A complete list of available functions can be found by typing **show functions** at the Asterisk command-line interface.

Conditional Branching

Now that you've learned a bit about expressions and functions, it's time to put them to use. By using expressions and functions, you can add even more advanced logic to your dialplan. To allow your dialplan to make decisions, you'll use *conditional branching*. Let's take a closer look.

The GotoIf( ) Application

The key to conditional branching is the GotoIf( ) application. GotoIf( ) evaluates an expression and sends the caller to a specific destination based on whether the expression evaluates to true or false.

GotoIf( ) uses a special syntax, often called the *conditional syntax*:

```
GotoIf(expression?destination1:destination2)
```

If the expression evaluates to true, the caller is sent to the first destination. If the expression evaluates to false, the caller is sent to the second destination. So, what is true and what is false? An empty string and the number 0 evaluate as false. Anything else evaluates as true.

The destinations can each be one of the following:

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

Either of the destinations may be omitted, but not both. If the omitted destination is to be followed, Asterisk simply goes on to the next priority in the current extension.

Let's use `GotoIf( )` in an example:

```
exten => 345,1,Set(TEST=1)
exten => 345,2,GotoIf($[{$TEST} = 1]?10:20)
exten => 345,10,Playback(weasels-eaten-phonesys)
exten => 345,20,Playback(office-iguanas)
```

By changing the value assigned to `TEST` in the first line, you should be able to have your Asterisk server play a different greeting.

Let's look at another example of conditional branching. This time, we'll use both `Goto( )` and `GotoIf( )` to count down from 10 and then hang up:

```
exten => 123,1,Set(COUNT=10)
exten => 123,2,GotoIf($[${COUNT} > 0]?:10)
exten => 123,3,SayNumber(${COUNT})
exten => 123,4,Set(COUNT=$[${COUNT} - 1])
exten => 123,5,Goto(2)
exten => 123,10,Hangup( )
```

Let's analyze this example. In the first priority, we set the variable `COUNT` to `10`. Next, we check to see if `COUNT` is greater than `0`. If it is, we move on to the next priority. (Don't forget that if we omit a destination in the `GotoIf( )` application, control goes to the next priority.) From there we speak the number, subtract `1` from `COUNT`, and go back to priority two. If `COUNT` is less than or equal to `0`, control goes to priority 10, and the call is hung up.

The classic example of conditional branching is affectionately known as the anti-girlfriend logic. If the Caller ID number of the incoming call matches the phone number of the recipient's ex-girlfriend, Asterisk gives a different message than it ordinarily would to any other caller. While somewhat simple and primitive, it's a good example for learning about conditional branching within the Asterisk dialplan.

This example uses a channel variable called `CALLERIDNUM`, which is automatically set by Asterisk to the Caller ID number of the inbound call. Let's assume for the sake of this example that the victim's phone number is 885-555-1212:

```
exten => 123,1,GotoIf($[${CALLERIDNUM} = 8885551212]?20:10)
exten => 123,10,Dial(Zap/4)
exten => 123,20,Playback(abandon-all-hope)
exten => 123,21,Hangup( )
```

In priority one, we call the `GotoIf( )` application. It tells Asterisk to go to priority 20 if the Caller ID number matches `8885551212`, and otherwise to go to priority 10. If the Caller ID number matches, control of the call goes to priority 20, which plays back an uninspiring message to the undesired caller. Otherwise, the call attempts to dial the recipient on channel `Zap/4`.

Time-Based Conditional Branching with GotoIfTime( )

Another way to use conditional branches in your dialplan is with the `GotoIfTime( )` application. Whereas `GotoIf( )` evaluates an expression to decide what to do, `GotoIfTime( )` looks at the current system time and uses that to decide whether or not to follow a different branch in the dialplan.

The most obvious use of this application is to give your callers a different greeting before and after normal business hours.

The syntax for the `GotoIfTime( )` application looks like this:

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

In short, `GotoIfTime( )` sends the call to the specified *`label`* if the current date and time match the criteria specified by *`times`*, *`days_of_week`*, *`days_of_month`*, and *`months`*. Let's look at each argument in more detail:

`times`
: This is a list of one or more time ranges, in 24-hour format. As an example, 9:00 a.m. through 5:00 p.m. would be specified as `09:00-17:00`. The day starts at 0:00 and ends at 23:59.

`days_of_week`
: This is a list of one or more days of the week. The days should be specified as `mon`, `tue`, `wed`, `thu`, `fri`, `sat`, and/or `sun`. Monday through Friday would be expressed as `mon-fri`. Tuesday and Thursday would be expressed as `tue,thu`.

`days_of_month`
: This is a list of the numerical days of the month. Days are specified by the numbers `1` through `31`. The 7th through the 12th would be expressed as `7-12`, and the 15th and 30th of the month would be written as `15,30`.

`months`
: This is a list of one or more months of the year. The months should be written as `jan`, `feb`, `mar`, `apr`, and so on.

If you wish to match on all possible values for any of these arguments, simply put an * in for that argument.

The *`label`* argument can be any of the following:

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

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

Now that we've covered the syntax, let's look at a couple of examples. The following example would match from *9:00 a.m. to 5:59 p.m.*, on *Monday through Friday*, on *any day of the month*, in *any month of the year*:

```
exten => s,1,GotoIfTime(09:00-17:59,mon-fri,*,*?open,s,1)
```

If the caller calls during these hours, the call will be sent to the first priority of the `s` extension in the context named `open`. If the call is made outside of the specified times, it will be sent to the next priority of the current extension. This allows you to easily branch on multiple times, as shown in the next example (note that you should always put your most specific time matches before the least specific ones):

```
; If it's any hour of the day, on any day of the week,
; during the fourth day of the month, in the month of of July,
; we're closed
exten => s,1,GotoIfTime(*,*,4,jul?open,s,1)

; During business hours, send calls to the open context
exten => s,2,GotoIfTime(09:00-17:59|mon-fri|*|*?open,s,1)
exten => s,3,GotoIfTime(09:00-11:59|sat|*|*?open,s,1)

; Otherwise, we're closed
exten => s,4,Goto(closed,s,1)
```

Voicemail

One of the most popular (or, actually, unpopular) features of any modern telephone system is voicemail. Naturally, Asterisk has a very flexible voicemail system. Some of the features of Asterisk's voicemail system include:

- Unlimited password-protected voicemail boxes, each containing mailbox folders for organizing voicemail
- Different greetings for busy and unavailable states
- Default and custom greetings
- The ability to associate phones with more than one mailbox and mailboxes with more than one phone
- Email notification of voicemail, with the voicemail optionally attached as a sound file[*]
- Voicemail forwarding and broadcasts

[*] No, you really don't have to pay for this; and yes, it really does work.

- Message-waiting indicator (flashing light or stuttered dial tone) on many types of phones
- Company directory of employees, based on voicemail boxes

And that's just the tip of the iceberg! In this section, we'll introduce you to the fundamentals of a typical voicemail setup.

The voicemail configuration is defined in the configuration file called *voicemail.conf*. This file contains an assortment of settings that you can use to customize the voicemail system to your needs. Covering all the available options in *voicemail.conf* would be beyond the scope of this chapter, but the sample configuration file is well documented and quite easy to follow. For now, look near the bottom of the file, where voicemail contexts and voicemail boxes are defined.

Just as dialplan contexts keep different parts of your dialplan separate, voicemail contexts allow you to define different sets of mailboxes that are separate from one another. This allows you to host voicemail for several different companies or offices on the same server. Voicemail contexts are defined in the same way as dialplan contexts, with square brackets surrounding the name of the context. For our examples, we'll be using the `[default]` voicemail context.

Creating Mailboxes

Inside each voicemail context, we define different mailboxes. The syntax for defining a mailbox is:

```
mailbox => password,name[,email[,pager_email[,options]]]
```

Let's explain what each part of the mailbox definition does:

`mailbox`
: This is the mailbox number. It usually corresponds with the extension number of the associated set.

`password`
: This is the numeric password the mailbox owner will use to access her voicemail. If the user changes her password, the system will update this field in the *voicemail.conf* file.

`name`
: This is the name of the mailbox owner. The company directory uses the text in this field to allow callers to spell usernames.

`email`
: This is the email address of the mailbox owner. Asterisk can send voicemail notifications (including the voicemail message itself) to the specified email box.

pager_email
: This is the email address of the mailbox owner's pager or cell phone. Asterisk can send a short voicemail notification message to the specified email address.

options
: This field is a list of options that sets the mailbox owner's time zone and overrides the global voicemail settings. There are nine valid options: `attach`, `serveremail`, `tz`, `saycid`, `review`, `operator`, `callback`, `dialout`, and `exitcontext`. These options should be in *option*=*value* pairs, separated by the pipe character (|). The `tz` option sets the user's time zone to a time zone previously defined in the `[zonemessages]` section of *voicemail.conf*, and the other eight options override the global voicemail settings with the same names.

A typical mailbox definition might look something like this:

```
101 => 1234,Joe Public,jpublic@somedomain.com,jpublic@pagergateway.net,tz=central|attach=yes
```

Continuing with our dialplan from the last chapter, let's set up voicemail boxes for John and Jane. We'll give John a password of `1234` and Jane a password of `4444` (remember, these go in *voicemail.conf*, not *extensions.conf*):

```
[default]
101 => 1234,John Doe,john@asteriskdocs.org,jdoe@pagergateway.tld
102 => 4444,Jane Doe,jane@asteriskdocs.org,jane@pagergateway.tld
```

Adding Voicemail to the Dialplan

Now that we've created mailboxes for Jane and John, let's allow callers to leave messages for them if they don't answer the phone. To do this, we'll use the `VoiceMail()` application.

The `VoiceMail()` application sends the caller to the specified mailbox, so that he can leave a message. The mailbox should be specified as *mailbox@context*, where *context* is the name of the voicemail context. The mailbox number can also be prefixed by the letter `b` or the letter `u`. If the letter `b` is used, the caller will hear the mailbox owner's *busy* message. If the letter `u` is used, the caller will hear the mailbox owner's *unavailable* message (if one exists).

Let's use this in our sample dialplan. Previously, we had a line like this in our `[internal]` context, which allowed us to call John:

```
exten => 101,1,Dial(${JOHN},,r)
```

Now, let's change it so that if John is busy (on another call), it'll send us to his voicemail, where we'll hear his busy message (don't forget that the `Dial()` application sends the caller to priority `n`+101 if the dialed line is busy):

```
exten => 101,1,Dial(${JOHN},,r)
exten => 101,102,VoiceMail(b101@default)
```

Next, let's add an unavailable message that the caller will be played if John doesn't answer the phone within 10 seconds. Remember, the second argument to the `Dial()` application is a timeout. If the call is not answered before the timeout expires, the call is sent to the next priority. Let's add a 10-second timeout, and a priority to send the caller to voicemail if John doesn't answer in time:

```
exten => 101,1,Dial(${JOHN},10,r)
exten => 101,2,VoiceMail(u101@default)
exten => 101,102,VoiceMail(b101@default)
```

If we add these two new priorities and a timeout argument to the `Dial()` application, callers will get John's voicemail (with the appropriate greeting) if John is either busy or unavailable. A slight problem remains, however, in that John has no way of retrieving his messages. Let's remedy that.

Accessing Voicemail

Users can retrieve their voicemail messages, change their voicemail options, and record their voicemail greetings by using the `VoiceMailMain()` application. In its typical form, `VoiceMailMain()` is called without any arguments. Let's add extension 500 to the `[internal]` context of our dialplan so that internal users can dial it to access their voicemail messages:

```
exten => 500,1,VoiceMailMain()
```

Creating a Dial-by-Name Directory

One last feature of the Asterisk voicemail system we should cover is the dial-by-name directory. This is created with the `Directory()` application. This application uses the names defined in the mailboxes in *voicemail.conf* to present the caller with a dial-by-name directory of the users.

`Directory()` takes up to three arguments: the voicemail context from which to read the names, the optional dialplan context in which to dial the user, and an option string (which is also optional). By default, `Directory()` searches for the user by last name, but passing the `f` option forces it to search by first name instead. Let's add two dial-by-name directories to the `[incoming]` context of our sample dialplan, so that callers can search by either first or last name:

```
exten => 8,1,Directory(default,incoming,f)
exten => 9,1,Directory(default,incoming)
```

If callers press 8, they'll get a directory by first name. If they dial 9, they'll get the directory by last name.

Macros

Macros are a very useful construct designed to avoid repetition in the dialplan. They also help in making changes to the dialplan. To illustrate this point, let's look at our sample dialplan again. If you remember the changes we made for voicemail, we ended up with the following for John's extension:

```
exten => 101,1,Dial(${JOHN},10,r)
exten => 101,2,VoiceMail(u101@default)
exten => 101,102,VoiceMail(b101@default)
```

Now imagine you have a hundred users on your Asterisk system—setting up the extensions would involve a lot of copying and pasting. Then imagine that you need to make a change to the way your extensions work. That would involve a lot of editing, and you'd be almost certain to have errors.

Instead, you can define a macro that contains a list of steps to take, and then have all of the phone extensions refer to that macro. All you need to change is the macro, and everything in the dialplan that references that macro will change as well.

If you're familiar with computer programming, you'll recognize that macros are similar to subroutines in many modern programming languages. If you're not familiar with computer programming, don't worry—we'll walk you through creating a macro.

The best way to appreciate macros is to see one in action, so let's move right along.

Defining Macros

For our first macro, let's take the dialplan logic we used above to set up voicemail for John and turn it into a macro. Then we'll use the macro to give John and Jane (and the rest of their coworkers) the same functionality.

Macro definitions look a lot like contexts. (In fact, you could argue that they really are small, limited contexts.) You define a macro by placing `macro-` and the name of your macro in square brackets, like this:

```
[macro-voicemail]
```

Macro names must start with `macro-`. This distinguishes them from regular contexts. The commands within the macro are built pretty nearly identically to anything else in the dialplan—the only limiting factor is that macros use only the `s` extension. Let's add our voicemail logic to the macro, changing the extension to `s` as we go:

```
[macro-voicemail]
exten => s,1,Dial(${JOHN},10,r)
exten => s,2,VoiceMail(u101@default)
exten => s,102,VoiceMail(b101@default)
```

That's a start, but it's not perfect, as it's still specific to John and his mailbox number. To make the macro generic so that it will work not only for John but also for all his coworkers, we'll take advantage of another property of macros: arguments. But first, let's see how we call macros in our dialplan.

Calling Macros from the Dialplan

To use a macro in our dialplan, we use the `Macro()` application. This application calls the specified macro and passes it any arguments. For example, to call our voicemail macro from our dialplan, we can do the following:

```
exten => 101,1,Macro(voicemail)
```

The `Macro()` application also defines several special variables for our use. They include:

`${MACRO_CONTEXT}`

The original context in which the macro was called.

`${MACRO_EXTEN}`

The original extension in which the macro was called.

`${MACRO_PRIORITY}`

The original priority in which the macro was called.

`${ARG`*`n`*`}`

The *n*th argument passed to the macro. For example, the first argument would be `${ARG1}`, the second `${ARG2}`, and so on.

As we explained earlier, the way we initially defined our macro was hard-coded for John, instead of being generic. Let's change our macro to use `${MACRO_EXTEN}` instead of `101` for the mailbox number. That way, if we call the macro from extension 101 the voicemail messages will go to mailbox 101, if we call the macro from extension 102 messages will go to mailbox 102, and so on:

```
[macro-voicemail]
exten => s,1,Dial(${JOHN},10,r)
exten => s,2,VoiceMail(u${MACRO_EXTEN}@default)
exten => s,102,VoiceMail(b${MACRO_EXTEN}@default)
```

Using Arguments in Macros

Okay, now we're getting closer to having the macro the way we want it, but we still have one thing left to change—we need to pass in the channel to dial, as it's currently still hard-coded for `${JOHN}` (remember that we defined the variable `JOHN` as the channel to call when we want to reach John). Let's pass in the channel as an argument, and then our first macro will be complete:

```
[macro-voicemail]
exten => s,1,Dial(${ARG1},10,r)
exten => s,2,VoiceMail(u${MACRO_EXTEN}@default)
exten => s,102,VoiceMail(b${MACRO_EXTEN}@default)
```

Now that our macro is done, we can use it in our dialplan. Here's how we can call our macro to provide voicemail to John, Jane, and Jack:

```
exten => 101,1,Macro(voicemail,${JOHN})
exten => 102,1,Macro(voicemail,${JANE})
exten => 103,1,Macro(voicemail,${JACK})
```

With 50 or more users, this dialplan will still look neat and organized—we'll simply have one line per user, referencing a macro that can be as complicated as required. We could even have a few different macros for various user types, such as `executives`, `courtesy_phones`, `call_center_agents`, `analog_sets`, `sales_department`, and so on.

A more advanced version of the macro might look something like this:

```
[macro-voicemail]
exten => s,1,Dial(${ARG1},20)
exten => s,2,Goto(s-${DIALSTATUS},1)
exten => s-NOANSWER,1,Voicemail(u${MACRO_EXTEN})
exten => s-NOANSWER,2,Goto(incoming,s,1)
exten => s-BUSY,1,Voicemail(b${MACRO_EXTEN})
exten => s-BUSY,2,Goto(incoming,s,1)
exten => _s-.,1,Goto(s-NOANSWER,1)
```

This macro depends on a nice side effect of the `Dial()` application: when you use the `Dial()` application, it sets the `DIALSTATUS` variable to indicate whether the call was successful or not. In this case, we're handling the `NOANSWER` and `BUSY` cases, and treating all other result codes as a `NOANSWER`.

Using the Asterisk Database (AstDB)

Having fun yet? It gets even better!

Asterisk provides a powerful mechanism for storing values, called the Asterisk database (AstDB). The AstDB provides a simple way to store data for use within your dialplan.

For those of you with experience using relational databases such as PostgreSQL or MySQL, the Asterisk database is not a traditional relational database. It is a Berkeley DB Version 1 database. There are several ways to store data from Asterisk in a relational database, but this book will not delve into them.

The Asterisk database stores its data in groupings called *families*, with values identified by *keys*. Within a family, a key may be used only once. For example, if we had a family called `test`, we could store only one value with a key called `count`. Each stored value must be associated with a family.

Storing Data in the AstDB

To store a new value in the Asterisk database, we use the `Set()` application,* but instead of using it to set a channel variable, we use it to set an AstDB variable. For example, to assign the `count` key in the `test` family the value of `1`:

```
exten => 456,1,Set(${DB(test/count)=1})
```

If a key named `count` already exists in the `test` family, its value will be overwritten with the new value. You can also store values from the Asterisk command line, by running the command **`database put`** ***`family key value`***. For our example, you would type **`database put test count 1`**.

Retrieving Data from the AstDB

To retrieve a value from the Asterisk database and assign it to a variable, we use the `Set()` application again. Let's retrieve the value of `count` (again, from the `test` family), assign it to a variable called `COUNT`, and then speak the value to the caller:

```
exten => 456,1,Set(DB(test/count)=1)
exten => 456,2,Set(COUNT=${DB(test/count)})
exten => 456,3,SayNumber(${COUNT})
```

You may also check the value of a given key from the Asterisk command line by running the command **`database get`** ***`family key`***. To view the entire contents of the AstDB, use the **`database show`** command.

Deleting Data from the AstDB

There are two ways to delete data from the Asterisk database. To delete a key, use the `DBdel()` application. It takes the family and key as arguments, like this:

```
exten => 457,1,DBdel(test/count)
```

You can also delete an entire key family by using the `DBdeltree()` application. The `DBdeltree()` application takes a single argument: the name of the key family to delete. To delete the entire `test` family, do the following:

```
exten => 457,1,DBdeltree(test)
```

* Previous versions of Asterisk had applications called `DBput()` and `DBget()` that were used to set values in and retrieve values from the AstDB. If you're using an old version of Asterisk, you'll want to use them instead.

To delete keys and key families from the AstDB via the command-line interface, use the **`database del`** ***`key`*** and **`database deltree`** ***`family`*** commands, respectively.

Using the AstDB in the Dialplan

There are an infinite number of ways to use the Asterisk database in a dialplan. To introduce the AstDB, we'll show two simple examples. The first is a simple counting example to show that the Asterisk database is persistent (meaning that it survives system reboots). In the second example, we'll use the `LookupBlacklist()` application to evaluate whether or not a number is on the blacklist and should be blocked.

To begin the counting example, let's first retrieve a number (the value of the `count` key) from the database and assign it to a variable named `COUNT`. If the key doesn't exist, `DBget()` will send us to priority `n+101`, where we will set the value to `1`. The next priority will send us back to priority 1. This will happen the very first time we dial this extension:

```
exten => 678,1,Set(COUNT=${DB(test/count)})
exten => 678,102,Set(DB(test/count)=1)
exten => 678,103,Goto(1)
```

Next, we'll say the current value of `COUNT`, and then increment `COUNT`:

```
exten => 678,1,Set(COUNT=${DB(test/count)})
exten => 678,2,SayNumber(${COUNT})
exten => 678,3,Set(COUNT=$[${COUNT} + 1])
exten => 678,102,Set(DB(test/count)=1)
exten => 678,103,Goto(1)
```

Now that we've incremented `COUNT`, let's put the new value back into the database. Remember that storing a value for an existing key overwrites the previous value:

```
exten => 678,1,Set(COUNT=${DB(test/count)})
exten => 678,2,SayNumber(${COUNT})
exten => 678,3,Set(COUNT=$[${COUNT} + 1])
exten => 678,4,Set(DB(test/count)=${COUNT})
exten => 678,102,Set(DB(test/count)=1)
exten => 678,103,Goto(1)
```

Finally, we'll loop back to the first priority. This way, the application will continue counting:

```
exten => 678,1,Set(COUNT=${DB(test/count)})
exten => 678,2,SayNumber(${COUNT})
exten => 678,3,SetVar(COUNT=$[${COUNT} + 1]
exten => 678,4,Set(DB(test/count)=${COUNT})
exten => 678,5,Goto(1)
exten => 678,102,Set(DB(test/count)=1)
exten => 678,103,Goto(1)
```

Go ahead and try this example. Listen to it count for a while, and then hang up. When you dial this extension again, it should continue counting from where it left off. The value stored in the database will be persistent, even across a restart of Asterisk.

In the next example, we'll create dialplan logic around the `LookupBlacklist( )` application, which checks to see if the current Caller ID number exists in the blacklist. (The blacklist is simply a family called `blacklist` in the AstDB.) If `LookupBlacklist( )` finds the number in the blacklist, it sends the call to priority `n+101`. Otherwise, the call continues on with the next priority:

```
exten => 124,1,LookupBlacklist( )
exten => 124,2,Dial(${JOHN})
exten => 124,102,Playback(privacy-you-are-blacklisted)
exten => 124,103,Playback(vm-goodbye)
exten => 124,104,Hangup( )
```

To add a number to the blacklist, run the **`database put blacklist`** ***`number`*** **`1`** command from the Asterisk command-line interface.

Handy Asterisk Features

Now that we've gone over some more of the basics, let's look at a few popular functions that have been incorporated into Asterisk.

Zapateller( )

`Zapateller( )` is a simple Asterisk application that plays a special information tone at the beginning of a call, which causes auto-dialers (usually used by telemarketers) to think the line has been disconnected. Not only will they hang up, but their systems will flag your number as out of service, which could help you avoid all kinds of telemarketing calls. To use this functionality within your dialplan, simply call the `Zapateller( )` application.

We'll also use the optional `nocallerid` option so that the tone will be played only when there is no Caller ID information on the incoming call. For example, you might use `Zapateller( )` in the `s` extension of your `[incoming]` context, like this:

```
[incomimg]
exten => s,1,Zapateller(nocallerid)
exten => s,2,Playback(enter-ext-of-person)
```

Call Parking

Another handy feature is called "call parking." Call parking allows you to place a call on hold in a "parking lot," so it can be taken off hold from another extension. Parameters for call parking (such as the extensions to use, the number of spaces, and so on) are all controlled within the *features.conf* configuration file. The `[general]` section of the *features.conf* file contains four settings related to call parking:

`parkext`
: This is the parking lot extension. Transfer a call to this extension, and the system will tell you which parking position the call is in. By default, the parking extension is `700`.

`parkpos`
: This option defines the number of parking slots. For example, setting it to `701-720` creates 20 parking positions, numbered 701 through 720.

`context`
: This is the name of the parking context. To be able to park calls, you must include this context.

`parkingtime`
: If set, this option controls how long (in seconds) a call can stay in the parking lot. If the call isn't picked up within the specified time, the extension that parked the call will be called back.

You must restart Asterisk after editing *features.conf*, as the file is read only on startup. Running the `reload` command will not cause the *features.conf* file to be read.

Also note that because the user needs to be able to transfer the calls to the parking lot extension, you should make sure you're using the `t` and/or `T` options to the `Dial()` application.

So, let's create a simple dialplan to show off call parking:

```
[incoming]
include => parkedcalls

exten=103,1,Dial(SIP/Bob,,tT)
exten=104,1,Dial(SIP/Charlie,,tT)
```

To illustrate how call parking works, say that Alice calls into the system and dials extension 103, to reach Bob. After a while, Bob transfers the call to extension 700, which tells him that the call from Alice has been parked in position 701. Bob then dials Charlie at extension 104, and tells him that Alice is at extension 701. Charlie then dials extension 701, and begins to talk to Alice. This is a simple and effective way of allowing callers to be transferred between users.

Conferencing with MeetMe( )

Last but not least, let's cover setting up an audio conference bridge with the `MeetMe( )` application.* This application allows multiple callers to converse together, as if they were all in the same physical location. Some of the main features include:

- The ability to create password-protected conferences
- Conference administration (mute conference, lock conference, kick participants)
- The option of muting all but one participant (useful for company announcements, broadcasts, etc.)
- Static or dynamic conference creation

Let's walk through setting up a basic conference room. The configuration options for the MeetMe conferencing system are found in *meetme.conf*. Inside the configuration file, you define conference rooms and optional numeric passwords. (If a password is defined here, it will be required to enter all conferences using that room.) For our example, let's set up a conference room at extension 600. First, we'll set up the conference room in *meetme.conf*. We'll call it `600`, and we won't assign a password at this time:

```
[rooms]
conf => 600
```

Now that the configuration file is complete, we'll need to restart Asterisk so that it can re-read the *meetme.conf* file. Next, we'll add support for the conference room to our dialplan with the `MeetMe( )` application. `MeetMe( )` takes three arguments: the name of the conference room (as defined in *meetme.conf*), a set of options, and the password the user must enter to join this conference. Let's set up a simple conference using room `600`, the `i` option (which announces when people enter and exit the conference), and a password of `54321`:

```
exten => 600,1,MeetMe(600,i,54321)
```

That's all there is to it! When callers enter extension 600, they will be prompted for the password. If they correctly enter `54321`, they will be added to the conference. See Appendix B for a list of all the options supported by the `MeetMe( )` application.

Another useful application is `MeetMeCount( )`. As its name suggests, this application counts the number of users in a particular conference room. It takes up to two arguments: the conference room in which to count the number of participants, and optionally a variable name to assign the count to. If the variable name is not passed as the second argument, the count is read to the caller:

```
exten => 601,1,Playback(conf-thereare)
exten => 601,2,MeetMeCount(600)
exten => 601,3,Playback(conf-peopleinconf)
```

* In the world of legacy PBXs, this type of functionality is very expensive. Either you have to pay big bucks for a dial-in service, or you have to add an expensive conferencing bridge to your proprietary PBX.

If you pass a variable as the second argument to `MeetMeCount( )`, the count is assigned to the variable and playback of the count is skipped. You might use this to limit the number of participants, like this:

```
; limit the conference room to 10 participants
exten => 600,1,MeetMeCount(600,CONFCOUNT)
exten => 600,2,GotoIf($[${CONFCOUNT} <= 10]?3:100)
exten => 600,3,MeetMe(600,i,54321)
exten => 600,100,Playback(conf-full)
```

Isn't Asterisk fun?

Conclusion

In this chapter, we've covered a few more of the many applications in the Asterisk dialplan, and hopefully we've given you the seeds from which you can explore the creation of your own dialplans. As with the previous chapter, we invite you to go back and re-read any sections that require clarification.

The following chapters take us away from Asterisk for a bit, in order to talk about some of the technologies that all telephone systems use. We'll be referring to Asterisk a lot, but much of what we want to discuss are things that are common to many telecom systems.