CHAPTER 5

Dialplan Basics

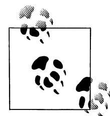
Everything should be made as simple as possible,
but not simpler.
—Albert Einstein (1879–1955)

The dialplan is truly the heart of any Asterisk system, as it defines how Asterisk handles inbound and outbound calls. In a nutshell, it consists of a list of instructions or steps that Asterisk will follow. Unlike traditional phone systems, Asterisk's dialplan is fully customizable. To successfully set up your own Asterisk system, you will need to understand the dialplan.

If writing a dialplan sounds overwhelming, don't worry. This chapter explains how dialplans work in a step-by-step manner and teaches the skills necessary to create your own. The examples have been designed to build upon one another, so feel free to go back and re-read a section if something doesn't quite make sense. Please also note that this chapter is by no means an exhaustive survey of all the possible things dialplans can do; our aim is to cover just the fundamentals. We'll cover more advanced dialplan topics in later chapters.

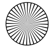
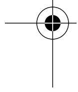
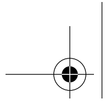
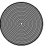
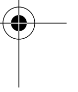
Dialplan Syntax

The Asterisk dialplan is specified in the configuration file named *extensions.conf*.



The *extensions.conf* file usually resides in the */etc/asterisk/* directory, but its location may vary depending on how you installed Asterisk. Other common locations for this file include */usr/local/asterisk/etc/* and */opt/asterisk/etc/*.

The dialplan is made up of four main parts: contexts, extensions, priorities, and applications. In the next few sections, we'll cover each of these parts and explain how they work together to create a dialplan. After explaining the role each of these

elements plays in the dialplan, we will step you though the process of creating a basic, functioning dialplan.

Sample Configuration Files

If you installed the sample configuration files when you installed Asterisk, you will most likely have an existing *extensions.conf* file. Instead of starting with the sample file, we suggest that you build your *extensions.conf* file from scratch. This will be very beneficial, as it will give you a better understanding of dialplan concepts and fundamentals.

That being said, the sample *extensions.conf* file remains a fantastic resource, full of examples and ideas that you can use after you've learned the basic concepts. We suggest you rename the sample file to something like *extensions.conf.sample*. That way, you can refer to it in the future. You can also find the sample configuration files in the */configs/* directory of the Asterisk source.

Contexts

Dialplans are broken into sections called *contexts*. Contexts are named groups of extensions. Simply put, they keep different parts of the dialplan from interacting with one another. An extension that is defined in one context is completely isolated from extensions in any another context, unless interaction is specifically allowed. (We'll cover how to allow interaction between contexts near the end of the chapter.)

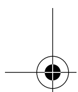
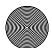
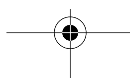
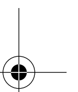
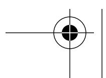
As a simple example, let's imagine we have two companies sharing an Asterisk server. If we place each company's voice menu in its own context, they are effectively separated from each other. This allows us to independently define what happens when, say, extension 0 is dialed: people pressing 0 at Company A's voice menu will get Company A's receptionist, and callers pressing 0 at Company B's voice menu will get Company B's receptionist. (This example assumes, of course, that we've told Asterisk to transfer the calls to the receptionists when callers press 0.)

Contexts are denoted by placing the name of the context inside square brackets ([]). The name can be made up of the letters A through Z (upper- and lowercase), the numbers 0 through 9, and the hyphen and underscore.* For example, a context for incoming calls looks like this:

```
[incoming]
```

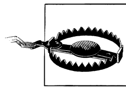
All of the instructions placed after a context definition are part of that context, until the next context is defined. At the beginning of the dialplan, there are two special

* Please note that the space is conspicuously absent from the list of allowed characters. Don't use spaces in your context names—you won't like the result!

contexts named [general] and [globals]. We will discuss the [globals] context later in this chapter; for now it's just important to know that these two contexts are special.

One of the most important uses of contexts is to enforce security. By using contexts correctly, you can give certain callers access to features (such as long-distance calling) that aren't made available to others. If you don't design your dialplan carefully, you may inadvertently allow others to fraudulently use your system. Please keep this in mind as you build your Asterisk system.



The Asterisk source contains a very important file named *SECURITY*, which outlines several steps you should take to keep your Asterisk system secure. It is vitally important that you read and understand this file. If you ignore the security precautions outlined there, you may end up allowing anyone and everyone to make long-distance or toll calls at your expense!

If you don't take the security of your Asterisk system seriously, you may end up paying—literally! *Please* take the time and effort to secure your system from toll fraud.

Extensions

Within each context, we define one or more extensions. An *extension* is an instruction that Asterisk will follow, triggered by an incoming call or by digits being dialed on a channel. Extensions specify what happens to calls as they make their way through the dialplan. Although extensions can be used to specify phone extensions in the traditional sense (i.e., please call John at extension 153), they can be used for much more in Asterisk.

The syntax for an extension is the word exten, followed by an arrow formed by the equals sign and the greater-than sign, like this:

```
exten =>
```

This is followed by the name of the extension. When dealing with telephone systems, we tend to think of extensions as the numbers you would dial to make another phone ring. In Asterisk, you get a whole lot more—for example, extension names can be any combination of numbers and letters. Over the course of this chapter and the next, we'll use both numeric and alphanumeric extensions.

Assigning names to extensions may seem like a revolutionary concept, but when you realize that many Voice-over-IP transports support (or even actively encourage) dialing by name or email address instead of by number, it makes perfect sense. This is one of the features that make Asterisk so flexible and powerful.

A complete extension is composed of three components:

- The name (or number) of the extension
- The priority (each extension can include multiple steps; the step number is called the "priority")
- The application (or command) that performs some action on the call

These three components are separated by commas, like this:

```
exten => name,priority,application()
```

Here's a simple example of what a real extension might look like:

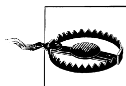
```
exten => 123,1,Answer()
```

In this example, the extension name is `123`, the priority is `1`, and the application is `Answer()`. Now, let's move ahead and explain priorities and applications.

Priorities

Each extension can have multiple steps, called *priorities*. Each priority is numbered sequentially, starting with 1. (Actually, there is one exception to this rule, as discussed in the sidebar "Unnumbered Priorities.") Each priority executes one specific application. As an example, the following extension would answer the phone (in priority number 1), and then hang it up (in priority number 2):

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



You must make sure that your priorities start at 1 and are numbered consecutively. If you skip a priority, Asterisk will not continue past it. If you find that Asterisk is not following all the priorities in a given extension, you may want to make sure you haven't skipped or misnumbered a priority.

Don't worry if you don't understand what `Answer()` and `Hangup()` are—we'll cover them shortly. The key point to remember here is that for a particular extension, Asterisk follows the priorities in numerical order.

Applications

Applications are the workhorses of the dialplan. Each application performs a specific action on the current channel, such as playing a sound, accepting touch-tone input, or hanging up the call. In the previous example, you were introduced to two simple applications: `Answer()` and `Hangup()`. You'll learn more about how these work momentarily.

Unnumbered Priorities

There's nothing like telling you that priorities have to be numbered sequentially, and then contradicting ourselves. Oh well, it needs to be done.

Version 1.2 of Asterisk adds a new twist to priority numbering. It introduces the use of the n priority, which stands for "next." Each time Asterisk encounters a priority named n, it takes the number of the previous priority and adds 1. This makes it easier to make changes to your dialplan, as you don't have to keep renumbering all your steps. For example, your dialplan might look something like this:

```
exten => 123,1,Answer()
exten => 123,n,do something
exten => 123,n,do something else
exten => 123,n,do one last thing
exten => 123,n,Hangup()
```

Version 1.2 also allows you to assign text labels to priorities. To assign a text label to a priority, simply add the label inside parentheses after the priority, like this:

```
exten => 123,n(label),do something
```

In the next chapter, we'll cover how to jump between different priorities based on dialplan logic.

Some applications, such as Answer() and Hangup(), need no other instructions to do their jobs. Other applications require additional information. These pieces of information, called *arguments*, can be passed on to the applications to affect how they perform their actions. To pass arguments to an application, place them between the parentheses that follow the application name, separated by commas.

Occasionally, you may also see the pipe character (|) being used as a separator between arguments, instead of a comma. Feel free to use whichever you prefer. For the examples in this book, however, we'll be using the comma to separate arguments to an application.

As we build our first dialplan in the next section, you'll learn to use applications (and their associated arguments) to your advantage.

A Simple Dialplan

Now we're ready to create our first dialplan. We'll start with a very simple example. We will design this dialplan so that as a call comes in, Asterisk will answer the call, play a sound file, and then hang up the call. We'll use this simple example to point out the most important dialplan fundamentals.

For the examples in this chapter to work correctly, we're assuming that at least one Zap channel has been created and configured (as described in the previous chapter), and that all incoming calls are sent to the `[incoming]` context. If you're using other types of channels, you may need to adjust these examples to fit your particular circumstances.

The s Extension

Before we get started with our dialplan, we ought to explain a special extension called `s`. When calls enter a context without a specific destination extension (for example, a ringing FXO line), they are handled automatically by the `s` extension. (The `s` stands for "start," as most calls start in the `s` extension.) Since this is exactly what we need for our dialplan, let's begin to fill in the pieces. We will be performing three actions on the call (answer it, play a sound file, and hang it up), so we need to create an extension called `s` with three priorities. We'll place the three priorities inside `[incoming]`, as all incoming calls should start in this context:

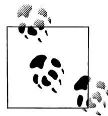
```
[incoming]
exten => s,1,application()
exten => s,2,application()
exten => s,3,application()
```

Now all we need to do is fill in the applications, and we've created our first dialplan.

The Answer( ), Playback( ), and Hangup( ) Applications

If we're going to answer the call, play a sound file, and then hang up, we'd better learn how to do just that. The `Answer()` application is used to answer a channel that is ringing. This does the initial setup for the channel that receives the incoming call. (A few applications don't require that you answer the channel first, but properly answering the channel before performing any other actions is a very good habit.) As we mentioned earlier, `Answer()` takes no arguments.

The `Playback()` application is used for playing a previously recorded sound file over a channel. When using the `Playback()` application, input from the user is simply ignored.



Asterisk comes with many professionally recorded sound files, which should be found in the default sounds directory (usually */var/lib/asterisk/sounds/*). They have been recorded in the GSM format, so they have a *.gsm* file extension. We'll be using these files in many of our examples. Several of the files in our examples come from the *asterisk-sounds* module, so please take the time to install it (see Chapter 3).

To use `Playback()`, specify a filename (without a file extension) as the argument. For example, `Playback(filename)` would play the sound file called *filename.gsm*,

assuming it was located in the default sounds directory. Note that you can include the full path to the file if you want, like this:

```
Playback(/home/john/sounds/filename)
```

This example would play *filename.gsm* from the */home/john/sounds/* directory. You can also use relative paths from the Asterisk sounds directory:

```
Playback(custom/filename)
```

This example would play *filename.gsm* from the *custom/* subdirectory of the default sounds directory. Note that if the specified directory contains more than one file with that filename but with different file extensions, Asterisk automatically plays the best file.*

The `Hangup()` application does exactly as its name implies: it hangs up the active channel. The caller will receive an indication that the call has been hung up. You will use this application at the end of a context when you want to end the current call, to ensure that callers don't continue on in the dialplan. This application takes no arguments.

Our First Dialplan

Now that we have created our extension, given it three different priorities, and learned about the applications we are going to use, let's put together all the pieces to create our first dialplan. As is typical in many technology books (especially computer programming books), our first example will be called "Hello World!"

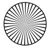
In the first priority of our extension, we'll answer the call. In the second, we'll play a sound file named *hello-world.gsm*, and in the third we'll hang up the call. Here's what the dialplan looks like:

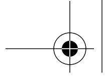
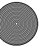
```
[incoming]
exten => s,1,Answer()
exten => s,2,Playback(hello-world)
exten => s,3,Hangup()
```

If you have a channel or two configured, go ahead and try it out! Simply make a new *extensions.conf* file with this short dialplan. (If it doesn't work, check the Asterisk console for error messages, and make sure your channels are configured to send inbound calls to the `[incoming]` context.)

Even though this example is very short and simple, it emphasizes the core concepts of contexts, extensions, priorities, and applications. Now that we've covered these

* Asterisk selects the best file based on translation cost; that is, it selects the file that is the least CPU-intensive to convert to its native audio format. When you start Asterisk, it calculates the translation costs between the different audio formats (they often vary from system to system). You can see these translation costs by typing **`show translation`** at the Asterisk command-line interface. We'll cover more about the different audio formats (known as *codecs*) in Chapter 8.



basic concepts, let's build upon our example. After all, a phone system that simply plays a sound file and then hangs up the channel isn't that useful!

Adding Logic to the Dialplan

The dialplan we just built was static—it always performs the same actions on every call. Now we'll start adding some logic to our dialplan so that it will perform different actions based on input from the user. We'll start by introducing a few more applications.

The Background( ) and Goto( ) Applications

One important key to building interactive Asterisk systems is the Background( ) application. Like Playback( ), it plays a recorded sound file. Unlike Playback( ), however, when the caller presses a key (or series of keys) on her telephone keypad, it interrupts the playback and goes to the extension that corresponds with the pressed digit(s). If a caller presses 5, for example, Asterisk will stop playing the sound file and send control of the call to the first priority of extension 5.

The most common use of the Background( ) application is to create voice menus (often called *auto-attendants* or *phone trees*). Many companies use voice menus to direct callers to the proper extensions, thus relieving their receptionists from having to answer every single call.

Background( ) has the same syntax as Playback( ):

```
exten => 123,1,Background(hello-world)
```

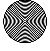
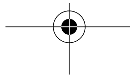
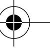
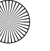
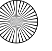
Another useful application is Goto( ). As its name implies, it is used to send the call to another context, extension, and priority. The Goto( ) application makes it easy to programmatically move a call between different parts of the dialplan. The syntax for the Goto( ) application calls for us to pass the destination context, extension, and priority as arguments to the application, like this:

```
exten => 123,1,Goto(context,extension,priority)
```

In our next example, we'll use the Background( ) and Goto( ) applications to create a slightly more complex dialplan, allowing the caller to interact with the system by pressing digits on the keypad. Let's begin by using Background( ) to accept input from the caller:

```
[incoming]
exten => s,1,Answer( )
exten => s,2,Background(enter-ext-of-person)
```

In this example, we'll play the sample sound file named *enter-ext-of-person.gsm*. While it's not the perfect fit for an auto-attendant greeting, it will certainly work for

this example. Now let's add two extensions that will be triggered by the caller entering either 1 or 2 at the prompt:

```
[incoming]
exten => s,1,Answer()
exten => s,2,Background(enter-ext-of-person)
exten => 1,1,Playback(digits/1)
exten => 2,1,Playback(digits/2)
```

Before going on, let's review what we've done so far. When users call into our dialplan, they will hear a greeting saying, "Please enter the number you wish to call." If they press 1, they will hear the number one, and if they press 2, they will hear the number two. While that's a good start, let's embellish it a little. We'll use the `Goto()` application to make the dialplan repeat the greeting after playing back the number:

```
[incoming]
exten => s,1,Answer()
exten => s,2,Background(enter-ext-of-person)
exten => 1,1,Playback(digits/1)
exten => 1,2,Goto(incoming,s,1)
exten => 2,1,Playback(digits/2)
exten => 2,2,Goto(incoming,s,1)
```

These two new lines (highlighted in bold) will send the call control back to the `s` extension after playing back the selected number.

If you look up the details of the `Goto()` application, you'll find that you can actually pass either one, two, or three arguments to the application. If you pass a single argument, it'll assume it's the destination priority in the current extension. If you pass two, it'll treat them as the extension and priority to go to in the current context.

In this example, we've passed all three arguments for the sake of clarity, but passing just the extension and priority would have had the same effect.

Handling Invalid Entries and Timeouts

Now that our first voice menu is fairly complete, let's add some additional special extensions. First, we need an extension for invalid entries, so that when a caller presses an invalid entry (e.g., pressing 3 in the above example), the call is sent to the `i` extension. Second, we need an extension to handle situations when the caller doesn't give input in time (the default timeout is 10 seconds). Calls will be sent to the `t` extension if the caller takes too long to press a digit after `Background()` has finished playing the sound file. Here is what our dialplan will look like after we've added these two extensions:

```
[incoming]
exten => s,1,Answer()
exten => s,2,Background(enter-ext-of-person)
exten => 1,1,Playback(digits/1)
```

```
exten => 1,2,Goto(incoming,s,1)
exten => 2,1,Playback(digits/2)
exten => 2,2,Goto(incoming,s,1)
exten => i,1,Playback(pbx-invalid)
exten => i,2,Goto(incoming,s,1)
exten => t,1,Playback(vm-goodbye)
exten => t,2,Hangup( )
```

Using the i and t extensions makes our dialplan a little more robust and user-friendly. That being said, it is still quite limited, because outside callers have no way of connecting to a live person. To do that, we'll need to learn about another application, called Dial( ).

Using the Dial( ) Application

One of Asterisk's most valuable features is its ability to connect different callers to each other. This is especially useful when callers are using different methods of communication. For example, caller A might be communicating over the standard analog telephone network, while user B might be sitting in a café halfway around the world and speaking on an IP telephone. Luckily, Asterisk takes most of the hard work out of connecting and translating between disparate networks. All you have to do is learn how to use the Dial( ) application.

The syntax of the Dial( ) application is a little more complex than that of the other applications we've used so far, but don't let that scare you off. Dial( ) takes up to four arguments. The first is the destination you're attempting to call, which is made up of a technology (or transport) across which to make the call, a forward slash, and the remote resource (usually a channel name or number). For example, let's assume that we want to call a Zap channel named Zap/1, which is an FXS channel with an analog phone plugged into it. The technology is "Zap," and the resource is "1." Similarly, a call to a SIP device might have a destination of SIP/1234, and a call to an IAX device might have a destination of IAX/fred. If we want Asterisk to ring the Zap/1 channel when extension 123 is reached in the dialplan, we'd add the following extension:

```
exten => 123,1,Dial(Zap/1)
```

When this extension is executed, Asterisk will ring the phone connected to channel Zap/1. If that phone is answered, Asterisk will bridge the inbound call with the Zap/1 channel. We can also dial multiple channels at the same time, by concatenating the destinations together with an ampersand (&), like this:

```
exten => 123,1,Dial(Zap/1&Zap/2&Zap/3)
```

The Dial( ) application will bridge the inbound call with whichever destination channel is answered first.

The second argument to the Dial( ) application is a timeout, specified in seconds. If a timeout is given, Dial( ) will attempt to call the destination(s) for that number of

seconds before giving up and moving on to the next priority in the extension. If no timeout is specified, `Dial()` will continue to dial the called channel(s) until someone answers or the caller hangs up. Let's add a timeout of 10 seconds to our extension:

```
exten => 123,1,Dial(Zap/1,10)
```

If the call is answered before the timeout, the channels are bridged and the dialplan is done. If the destination simply does not answer, `Dial()` goes on to the next priority in the extension. If, however, the destination channel is busy, `Dial()` will go to priority `n`+101, if it exists (where `n` is the priority where the `Dial()` application was called). This allows us to handle unanswered calls differently from calls whose destinations were busy.

Let's put what we've learned so far into another example:

```
exten => 123,1,Dial(Zap/1,10)
exten => 123,2,Playback(vm-nobodyavail)
exten => 123,3,Hangup( )
exten => 123,102,Playback(tt-allbusy)
exten => 123,103,Hangup( )
```

As you can see, this example will play the *vm-nobodyavail.gsm* sound file if the call goes unanswered, or the *tt-allbusy.gsm* sound file if the `Zap/1` channel is currently busy.

The third argument to `Dial()` is an option string. It may contain one or more characters that modify the behavior of the `Dial()` application. While the list of possible options is too long to cover here, the most popular option is the letter `r`. If you place the letter `r` as the third argument, the calling party will hear a ringing tone while the destination channel is being notified of an incoming call.

It should be noted that the `r` option isn't always required to indicate ringing, as Asterisk will automatically generate a ringing tone when it is attempting to establish a channel. However, you can use the `r` option to force Asterisk to indicate ringing even when no connection is being attempted. To add the `r` option to our last example, we simply change the first line:

```
exten => 123,1,Dial(Zap/1,10,r)
exten => 123,2,Playback(vm-nobodyavail)
exten => 123,3,Hangup( )
exten => 123,102,Playback(tt-allbusy)
exten => 123,103,Hangup( )
```

Since the extensions numbered 1 and 2 in our dialplan are somewhat useless now that we know how to use the `Dial()` application, let's replace them with extensions 101 and 102, which will allow outside callers to reach John and Jane:

```
[incoming]
exten => s,1,Answer( )
exten => s,2,Background(enter-ext-of-person)
exten => 101,1,Dial(Zap/1,10)
exten => 101,2,Playback(vm-nobodyavail)
```

```
exten => 101,3,Hangup( )
exten => 101,102,Playback(tt-allbusy)
exten => 101,103,Hangup( )
exten => 102,1,Dial(SIP/Jane,10)
exten => 102,2,Playback(vm-nobodyavail)
exten => 102,3,Hangup( )
exten => 102,102,Playback(tt-allbusy)
exten => 102,103,Hangup( )
exten => i,1,Playback(pbx-invalid)
exten => i,2,Goto(incoming,s,1)
exten => t,1,Playback(vm-goodbye)
exten => t,2,Hangup( )
```

The fourth and final argument to the `Dial( )` application is a URL. If the destination channel supports receiving a URL at the time of the call, the specified URL will be sent (for example, if you have an IP telephone that supports receiving a URL, it will appear on the phone's display; likewise, if you're using a soft phone, the URL might pop up on your computer screen). This argument is very rarely used.

If you are making outbound calls on an FXO Zap channel, you can use the following syntax to dial a number on that channel:

```
exten => 123,1,Dial(Zap/4/5551212)
```

This example would dial the number 555-1212 on the `Zap/4` channel. For other channel types, such as SIP and IAX, simply put the destination as the resource, as shown in these two lines:

```
exten => 123,1,Dial(SIP/1234)
exten => 124,1,Dial(IAX2/john@asteriskdocs.org)
```

Note that any of these arguments may be left blank. For example, if you want to specify an option but not a timeout, simply leave the timeout argument blank, like this:

```
exten => 123,1,Dial(Zap/1,,r)
```

Adding a Context for Internal Calls

In our examples thus far we have limited ourselves to a single context, but it is probably fair to assume that almost all Asterisk installations will have more than one context in their dialplans. As we mentioned at the beginning of this chapter, one important function of contexts is to separate privileges (such as making long-distance calls or calling certain extensions) for different classes of callers. In our next example, we'll add to our dialplan by creating two internal phone extensions, and we'll set up the ability for these two extensions to call each other. To accomplish this, we'll create a new context called `[internal]`.

As in previous examples, we've assumed that an FXS Zap channel (Zap/1, in this case) has already been configured, and that your *zapata.conf* file is configured so that any calls originated by Zap/1 begin in the [internal] context. For a few examples at the end of the chapter, we'll also assume that an FXO Zap channel has been configured as Zap/4, with calls coming in on this channel being sent to the [incoming] context. This channel will be used for outbound calling.

We've also assumed you have at least one SIP channel (named SIP/jane) that is configured to originate in the [internal] context. We've done this to introduce you to using other types of channels.

If you don't have hardware for the channels listed above (such as Zap/4), or if you're using hardware with different channel names (e.g., not SIP/jane), don't worry—you can change the examples to match your particular system configuration.

Our dialplan now looks like this:

```
[incoming]
exten => s,1,Answer( )
exten => s,2,Background(enter-ext-of-person)
exten => 101,1,Dial(Zap/1,10)
exten => 101,2,Playback(vm-nobodyavail)
exten => 101,3,Hangup( )
exten => 101,102,Playback(tt-allbusy)
exten => 101,103,Hangup( )
exten => 102,1,Dial(SIP/Jane,10)
exten => 102,2,Playback(vm-nobodyavail)
exten => 102,3,Hangup( )
exten => 102,102,Playback(tt-allbusy)
exten => 102,103,Hangup( )
exten => i,1,Playback(pbx-invalid)
exten => i,2,Goto(incoming,s,1)
exten => t,1,Playback(vm-goodbye)
exten => t,2,Hangup( )

[internal]
exten => 101,1,Dial(Zap/1,,r)
exten => 102,1,Dial(SIP/jane,,r)
```

In this example, we have added two new extensions to the [internal] context. This way, the person using channel Zap/1 can pick up the phone and dial the person at channel SIP/jane by dialing 102. By that same token, the phone registered as SIP/jane can dial Zap/1 by dialing 101.

We've arbitrarily decided to use extensions 101 and 102 for our examples, but feel free to use whatever numbering convention you wish for your extensions. You should also be aware that you're not limited to three-digit extensions—you can use as few or as many digits as you like. (Well, almost. Extensions must be shorter than

80 characters long, and you shouldn't use single-character extensions for your own use, as they're reserved.) Don't forget that you can use names as well, like so:

```
[incoming]
exten => s,1,Answer( )
exten => s,2,Background(enter-ext-of-person)
exten => 101,1,Dial(Zap/1,10)
exten => 101,2,Playback(vm-nobodyavail)
exten => 101,3,Hangup( )
exten => 101,102,Playback(tt-allbusy)
exten => 101,103,Hangup( )
exten => 102,1,Dial(SIP/Jane,10)
exten => 102,2,Playback(vm-nobodyavail)
exten => 102,3,Hangup( )
exten => 102,102,Playback(tt-allbusy)
exten => 102,103,Hangup( )
exten => t,1,Playback(vm-goodbye)
exten => t,2,Hangup( )

[internal]
exten => 101,1,Dial(Zap/1,,r)
exten => john,1,Dial(Zap/1,,r)
exten => 102,1,Dial(SIP/jane,,r)
exten => jane,1,Dial(SIP/jane,,r)
```

It certainly wouldn't hurt to add named extensions if you think your users might be dialed via a VoIP transport that supports names.

Now that our internal callers can call each other, we're well on our way toward having a complete dialplan. Next, we'll see how we can make our dialplan more scalable and easier to modify in the future.

Using Variables

Variables can be used in an Asterisk dialplan to help reduce typing, add clarity, or add additional logic to a dialplan. If you have some computer programming experience, you probably already understand what a variable is. If not, don't worry; we'll explain what variables are and how they are used.

You can think of a variable as a container that can hold one value at a time. So, for example, we might create a variable called `JOHN` and assign it the value of `Zap/1`. This way, when we're writing our dialplan, we can refer to John's channel by name, instead of remembering that John is using `Zap/1`. To assign a value to a variable, simply type the name of the variable, an equals sign, and the value, like this:

```
JOHN=Zap/1
```

There are two ways to reference a variable. To reference the variable's name, simply type the name of the variable, such as `JOHN`. If, on the other hand, you want to reference its value, you must type a dollar sign, an opening curly brace, the name of the

variable, and a closing curly brace. Here's how we'd reference the variable inside the Dial( ) application:

```
exten => 555,1,Dial(${JOHN},,r)
```

In our dialplan, whenever we write ${JOHN}, Asterisk will automatically replace it with whatever value has been assigned to the variable named JOHN.

Note that variable names don't have to be capitalized, but we're doing so in this book for readability's sake.

There are three types of variables we can use in our dialplan: global variables, channel variables, and environment variables. Let's take a moment to look at each type.

Global variables

As their name implies, *global* variables apply to all extensions in all contexts. Global variables are useful in that they can be used anywhere within a dialplan to increase readability and manageability. Suppose for a moment that you had a large dialplan and several hundred references to the Zap/1 channel. Now imagine you had to go through your dialplan and change all those references to Zap/2. It would be a long and error-prone process, to say the least.

On the other hand, if you had defined a global variable with the value Zap/1 at the beginning of your dialplan and then referenced that instead, you would only have to change one line.

Global variables should be declared in the [globals] context at the beginning of the *extensions.conf* file. They can also be defined programmatically, using the SetGlobalVar( ) application. Here is how both methods look inside of a dialplan:

```
[globals]
JOHN=Zap/1

[internal]
exten => 123,1,SetGlobalVar(JOHN=Zap/1)
```

Channel variables

A *channel* variable is a variable (such as the Caller*ID number) that is associated only with a particular call. Unlike global variables, channel variables are defined only for the duration of the current call and are available only to the channel participating in that call.

There are many predefined channel variables available for use within the dialplan, which are explained in the *README.variables* file in the *doc* subdirectory of the Asterisk source. Channel variables are set via the Set( ) application:

```
exten => 123,1,Set(MAGICNUMBER=42)
```

We'll use several of these channel variables in the next chapter.

Environment variables

Environment variables are a way of accessing Unix environment variables from within Asterisk. These are referenced in the form of `${ENV(var)}`, where *var* is the Unix environment variable you wish to reference.

Adding variables to our dialplan

Now that we've learned about variables, let's put them to work in our dialplan. We'll add variables for two people, John and Jane:

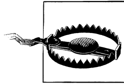
```
[globals]
JOHN=Zap/1
JANE=SIP/jane

[incoming]
exten => s,1,Answer( )
exten => s,2,Background(enter-ext-of-person)
exten => 101,1,Dial(${JOHN},10)
exten => 101,2,Playback(vm-nobodyavail)
exten => 101,3,Hangup( )
exten => 101,102,Playback(tt-allbusy)
exten => 101,103,Hangup( )
exten => 102,1,Dial(${JANE},10)
exten => 102,2,Playback(vm-nobodyavail)
exten => 102,3,Hangup( )
exten => 102,102,Playback(tt-allbusy)
exten => 102,103,Hangup( )
exten => i,1,Playback(pbx-invalid)
exten => i,2,Goto(incoming,s,1)
exten => t,1,Playback(vm-goodbye)
exten => t,2,Hangup( )

[internal]
exten => 101,1,Dial(${JOHN},,r)
exten => 102,1,Dial(${JANE},,r)
```

Pattern Matching

Often, it would be tedious to add every possible extension to a dialplan. This is especially the case for outbound calls. Can you imagine a dialplan with an extension for every possible number you could dial? Luckily, Asterisk has just the thing for situations like this: *pattern matching* to allow you to use one section of code for many different extensions.

Pattern-matching syntax

When using pattern matching, we use different letters and symbols to represent the possible digits we want to match. Patterns always start with an underscore (_). This tells Asterisk that we're matching on a pattern, and not on an extension name. (This means, of course, that you should never start your extension names with an underscore.)



If you forget the underscore on the front of your pattern, Asterisk will think it's just a named extension and won't do any pattern matching.

After the underscore, you can use one or more of the following characters:

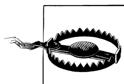
`X`
: Matches any digit from 0 to 9.

`Z`
: Matches any digit from 1 to 9.

`N`
: Matches any digit from 2 to 9.

`[15-7]`
: Matches any digit or range of digits specified. In this case, matches a 1, 5, 6, or 7.

`.` *(period)*
: Wildcard match; matches one or more characters.



If you're not careful, wildcard matches can make your dialplans do things you're not expecting. You should only use the wildcard match in a pattern after you've matched as many other digits as possible. For example, the following pattern match should probably never be used:

```
_.
```

In fact, Asterisk will warn you if you try to use it. Instead, use this one, if possible:

```
_X.
```

To use pattern matching in your dialplan, simply put the pattern in the place of the extension name (or number):

```
exten => _NXX,1,Playback(auth-thankyou)
```

In this example, the pattern would match any 3-digit extension from 200 through 999 (the `N` matches any digit between 2 and 9, and each `X` matches a digit between 0 and 9). That is to say, if a caller dialed any 3-digit extension between 200 and 999 in this context, he would hear the sound file *auth-thankyou.gsm*.

One other important thing to know about pattern matching is that if Asterisk finds more than one pattern that matches the dialed extension, it will use the *most specific* one. Say you had defined the following two patterns, and a caller dialed 888-555-1212:

```
exten => _555XXXX,1,Playback(digits/1)
exten => _55512XX,1,Playback(digits/2)
```

In this case the second extension would be selected, because it is more specific.

Pattern-matching examples

Before we go on, let's look at a few more pattern-matching examples. In each one, see if you can tell what the pattern would match before reading the explanation. We'll start with an easy one:

```
_NXXXXXX
```

Got it? This pattern would match any seven-digit number, as long as the first digit was two or higher. According to the North American Numbering Plan, this pattern would match any local number.

> ### The NANP and Toll Fraud
>
> The North American Number Plan (NANP) is a shared telephone numbering scheme used by 19 countries in North America and the Caribbean.
>
> In the United States and Canada, telecom regulations are similar (and sensible) enough that you can place a long-distance call to most numbers in country code 1 and expect to pay a reasonable toll. What many people don't realize, however, is that 19 countries, many of which have very different telecom regulations, share the NANP. (More information can be found at *http://www.nanpa.com*.)
>
> Many toll-fraud schemes trick naive North Americans into calling shockingly expensive per-minute toll numbers in a Caribbean country—the callers believe that since they dialed 1-NPA-NXX-XXXX to reach the number, they'll be paying their standard national long-distance rate for the call. Since the country in question may have regulations that allow for this form of extortion, the caller is ultimately held responsible for the call charges.
>
> The only way to prevent this sort of activity is to block calls to certain area codes (809, for example) and remove the restrictions only on an as-needed basis. Please take extra caution to make sure users can't abuse your phone system!

Let's try another:

```
_1NXXNXXXXXX
```

This one is slightly more difficult. This would match the number 1, followed by an area code between 200 and 999, then any 7-digit number. In the NANP, you would use this pattern to match any long-distance number.

Now for an even trickier example:

```
_011.
```

If that one left you scratching your head, look at it again. Did you notice the period on the end? This pattern matches any number that starts with 011 and has at least one more digit. In the NANP, this indicates an international phone number.*

Using the ${EXTEN} channel variable

We know what you're thinking... You're sitting there asking yourself, "So what happens if I want to use pattern matching, but I need to know which digits were actually dialed?" Luckily, Asterisk has just the answer. Whenever you dial an extension, Asterisk sets the `${EXTEN}` channel variable to the digits that were dialed. We can use an application called `SayDigits()` to test it out:

```
exten => _XXX,1,SayDigits(${EXTEN})
```

In this example, the `SayDigits()` application will read back to you the three-digit extension you dialed.

Often, it's useful to manipulate the `${EXTEN}` by stripping a certain number of digits off the front of the extension. This is accomplished by using the syntax `${EXTEN:x}`, where *x* is the number of digits you'd like to remove. For example, if the value of `EXTEN` is 95551212, `${EXTEN:1}` equals 5551212. Let's take a look at another example:

```
exten => _XXX,1,SayDigits(${EXTEN:1})
```

In this example, the `SayDigits()` application would read back only the last two digits of the dialed extension.

If *x* is negative, `SayDigits()` gives you the last *x* digits of the dialed extension. In this next example, `SayDigits()` will read back only the last digit of the dialed extension:

```
exten => _XXX,1,SayDigits(${EXTEN:-1})
```

Enabling Outbound Dialing

Now that we've introduced pattern matching, we can go about the process of allowing users to make outbound calls. The first thing we'll do is add a variable to the `[globals]` context to define which channel will be used for outbound calls:

```
[globals]
JOHN=Zap/1
JANE=SIP/jane
OUTBOUNDTRUNK=Zap/4
```

* If you find it peculiar that we've chosen patterns that are used to dial outbound numbers in the NANP, you're on to something! We'll be using these patterns in the next section to add outbound dialing capabilities to our dialplan.

Next, we will add contexts to our dialplan for outbound dialing.

You may be asking yourself at this point, "Why do we need separate contexts for outbound calls?" This is so that we can regulate and control who has permission to make outbound calls, and which types of outbound calls they are allowed to make.

First, let's make a context for local calls. To be consistent with most traditional phone switches, we'll put a 9 on the front of our patterns, so that users have to dial 9 before calling an outside number:

```
[outbound-local]
exten => _9NXXXXXX,1,Dial(${OUTBOUNDTRUNK}/${EXTEN:1})
exten => _9NXXXXXX,2,Congestion( )
exten => _9NXXXXXX,102,Congestion( )
```

Note that dialing 9 doesn't actually give you an outside line, unlike with many traditional PBX systems. Once you dial 9 on an FXS line, the dial tone will stop. If you'd like the dial tone to continue even after dialing 9, add the following line (right after your context definition):

```
ignorepat => 9
```

This directive tells Asterisk to continue to provide a dial tone, even after the caller has dialed the indicated pattern.

Let's review what we've just done. We've added a global variable called `OUTBOUNDTRUNK`, which will control which channel to use for outbound calls. We've also added a context for local outbound calls. In priority 1, we take the dialed extension, strip off the 9 with the `${EXTEN:1}` syntax, and then attempt to dial that number on the channel signified by the variable `OUTBOUNDTRUNK`. If the call is successful, the caller is bridged with the outbound channel. If the call is unsuccessful (because either the channel is busy or the number can't be dialed for some reason), the `Congestion( )` application is called, which plays a "fast busy signal" (congestion tone) to let the caller know that the call was unsuccessful.

Before we go any farther, let's make sure our dialplan allows outbound emergency numbers:

```
[outbound-local]
exten => _9NXXXXXX,1,Dial(${OUTBOUNDTRUNK}/${EXTEN:1})
exten => _9NXXXXXX,2,Congestion( )
exten => _9NXXXXXX,102,Congestion( )

exten => 911,1,Dial(${OUTBOUNDTRUNK}/911)
exten => 9911,1,Dial(${OUTBOUNDTRUNK}/911)
```

Again, we're assuming for the sake of these examples that we're inside the United States or Canada. If you're outside of this area, please replace 911 with the emergency services number in your particular location. This is something you never want to forget to put in your dialplan!

Next, let's add a context for long-distance calls:

```
[outbound-long-distance]
exten => _91NXXNXXXXXX,1,Dial(${OUTBOUNDTRUNK}/${EXTEN:1})
exten => _91NXXNXXXXXX,2,Congestion( )
exten => _91NXXNXXXXXX,102,Congestion( )
```

Now that we have these two new contexts, how do we allow internal users to take advantage of them? We need a way for contexts to be able to use other contexts.

Includes

Asterisk enables us to use a context within another context via the `include` directive. This is used to grant access to different sections of the dialplan. We'll use the include functionality to allow users in our `[internal]` context the ability to make outbound phone calls. But first, let's cover the syntax.

The `include` statement takes the following form, where *`context`* is the name of the remote context we want to include in the current context:

```
include => context
```

When we include other contexts within our current context, we have to be mindful of the order in which we including them. Asterisk will first try to match the extension in the current context. If unsuccessful, it will then try the first included context, and then continue to the other included contexts in the order in which they were included.

As it sits, our current dialplan has two contexts for outbound calls, but there's no way for people in the `[internal]` context to use them. Let's remedy that by including the two outbound contexts in the `[internal]` context, like this:

```
[globals]
JOHN=Zap/1
JANE=SIP/jane
OUTBOUNDTRUNK=Zap/4

[incoming]
exten => s,1,Answer( )
exten => s,2,Background(enter-ext-of-person)
exten => 101,1,Dial(${JOHN},10)
exten => 101,2,Playback(vm-nobodyavail)
exten => 101,3,Hangup( )
exten => 101,102,Playback(tt-allbusy)
exten => 101,103,Hangup( )
exten => 102,1,Dial(${JANE},10)
exten => 102,2,Playback(vm-nobodyavail)
exten => 102,3,Hangup( )
exten => 102,102,Playback(tt-allbusy)
exten => 102,103,Hangup( )
exten => i,1,Playback(pbx-invalid)
exten => i,2,Goto(incoming,s,1)
```

```
exten => t,1,Playback(vm-goodbye)
exten => t,2,Hangup( )

[internal]
include => outbound-local
include => outbound-long-distance

exten => 101,1,Dial(${JOHN},,r)
exten => 102,1,Dial(${JANE},,r)

[outbound-local]
exten => _9NXXXXXX,1,Dial(${OUTBOUNDTRUNK}/${EXTEN:1})
exten => _9NXXXXXX,2,Congestion( )
exten => _9NXXXXXX,102,Congestion( )

exten => 911,1,Dial(${OUTBOUNDTRUNK}/911)
exten => 9911,1,Dial(${OUTBOUNDTRUNK}/911)

[outbound-long-distance]
exten => _91NXXNXXXXXX,1,Dial(${OUTBOUNDTRUNK}/${EXTEN:1})
exten => _91NXXNXXXXXX,2,Congestion( )
exten => _91NXXNXXXXXX,102,Congestion( )
```

These two `include` statements make it possible for callers in the `[internal]` context to make outbound calls. We should also note that for security's sake you should always make sure that your `[inbound]` context never allows outbound dialing. (If by chance it did, people could dial into your system, and then make outbound toll calls that would be charged to you!)

Conclusion

And there we have it—a basic but functional dialplan. It's not exactly fully featured, but we've covered all of the fundamentals. In the following chapters, we'll continue to add features to this foundation.

If parts of this dialplan don't make sense, you may want to go back and re-read a section or two before continuing on to the next chapter. It's imperative that you understand these principles and how to apply them, or the following chapters will only confuse you more. And we don't want you to be confused!