The Asterisk Documentation Project

Volume One: An Introduction to Asterisk

Leif Madsen

Jared Smith

Jim Van Meggelen

Chris Tooley

The Asterisk Documentation Project: Volume One: An Introduction to Asterisk
by Leif Madsen, Jared Smith, Jim Van Meggelen, and Chris Tooley

A guide to the installation and basic use of Asterisk.

Revision History

Revision 0.1 $Date: 2004/09/19 16:59:59 $

Table of Contents

Preface

Chapter 1. Introduction

General Concept of Asterisk

Asterisk: A Telephony Revolution

Welcome to the wonderful world of Asterisk. You are about to discover the most powerful and popular open source PBX available.

Asterisk allows you to craft a telephony system to address your specific requirements. It does this by providing a library of basic telephony functions which you then use as script building-blocks. Calls into the system trigger these functions through digit patterns (referred to as extensions), giving you complete control of complex call routing concepts with relative ease. Common PBX functionality such as voicemail, call queuing, conferencing, music on hold and others are all included. But that's just the beginning. Asterisk is one of the few PBXs in existence that connects legacy telephony technologies such as PRI or Analog trunks through the same switching logic as state of the art VoIP interfaces such as IAX2, H.323 or SIP. This powerful yet simple core allows complex concepts in other systems to be deployed with ease in Asterisk. For example, building an IVR application or deploying CTI functionality can be done more inexpensively than with any other system. Why? Because with Asterisk, it's all built right in!

Perhaps Asterisk's most valuable asset is the open nature of the system. As with any open-source application, Asterisk can be further enhanced by the community of people who use it. This powerful concept ensures that Asterisk is prepared to mature in keeping with the demands of the industry. Go ahead and modify the source code to fit your needs, or, better still, contribute to an active and growing development community.

Because Asterisk is so powerful and flexible, in this book we can only begin to cover all the possible uses and configurations. We will therefore focus on the most commonly used features of the system and answer the most frequently asked questions. If you can work through the material we present here, you will be well on your way to becoming a competent Asterisk solutions developer.

The goals of this document

Asterisk has evolved beyond its roots as an open source experiment. A community of enthusiasts and developers, led by Mark Spencer, have provided a platform that has recently become very interesting to a much wider audience. People who appreciate the incredible potential of this platform are using it to solve a diverse and fascinating plethora of problems.

What new users seem to have the most trouble with, however, is getting a basic system up and running.

In this document, we will walk you through the process of selecting a platform, installing hardware, obtaining Asterisk, compiling it, configuring it, and starting it.

When we are done, you should have a basic but fully-functional Asterisk PBX, with working FXO, FXS, IAX and SIP connections, as well as functioning examples of the various things that Asterisk can do.

Will this solve the particular challenge you require Asterisk to solve for you? Probably not. But you *will* have a system from which you can begin your journey of discovery.

Learning Asterisk is much like learning a new programming language. Once you get the hang of the basics, the rest comes easy; with practice, and time.

Prerequisite Knowledge and Skills

Because of the nearly limitless flexibility of Asterisk, successfully configuring a system requires more than a passing familiarity with several technical concepts; most notably Linux Installation and Administration, as well as an understanding of Telephony. In this document, we will only be scraping the surface of these complex technologies as we discuss concepts relevant to Asterisk design, installation and administration. If you desire more knowledge on either of the aforementioned subjects, we have suggested several sources which will set you on the path towards enlightenment

Telephony

Asterisk is a PBX, and that means that the more Telecommunications knowledge you have, the easier Asterisk will be to learn. If you plan to use legacy PSTN circuits and telephones, you will want to understand the difference between FXS and FXO interfaces. Digital trunks will require you to be conversant with technologies such as ISDN-PRI (including wiring of T1s). Terms such as PSTN or VoIP should be familiar to you, and you'd do well to obtain an understanding of the concept of analog to digital conversion, and what codecs are.

Before you get overwhelmed, please understand that many excellent references exist to help you obtain this knowledge. A good introductory work is Noll's Introduction to Telephones and Telephone Systems , published by Artech House Publishers. The definitive encyclopaedia of all things Telecom is Newton's Telecom Dictionary , published by CMP Books - this book should be on any telecommunication professional's bookshelf.

Linux Installation and Administration

You will need an i386-compatible system with Linux installed before you can install and use Asterisk. If you do not have a solid grasp of Linux administration concepts you will want to focus on honing those skills before attempting an Asterisk install. Everyone has to crawl before we can walk, no? On the Internet, The Linux Documentation Project (http://www.tldp.org) provides many great resources for beginners. In the bookstore, Frisch's Essential System Administration along with Nemeth, et al.'s Linux Administration Handbook and Unix System Administration Handbook are recommended. Running Linux by Matt Welch, Lar Kaufman et al. is still one of the all-time most successful Linux introductions. Reading one or two of these books can save a lot of headaches down the road.

What to expect

Asterisk is not a turnkey system

The Asterisk PBX system is a complex piece of software. The learning curve is very steep and simply reading any single resource will not teach you everything that Asterisk is capable of. It's a safe bet that not even Mark Spencer knows all of the things his creation is capable of.

This book will attempt to address some of the most common issues that newcomers to Asterisk encounter, using step-by-step processes, ultimately resulting in the creation of a functioning Asterisk PBX system.

Learning how Asterisk works is very much like learning a new programming language. Many hours need to be spent with Asterisk in order to grasp how all the configuration files work with each other to control the many interfaces. We will be

glossing over many of these topics in pursuit of our more simple goal, but some things will need to be discussed and understood.

Comprehending the dialplan is a fundamental concept that those new to Asterisk must grasp. The configuration of the different types of channels Asterisk uses to communicate with ultimately are brought together in the dialplan. It is the heart of the Asterisk system.

Many people experience much pain and frustration when attempting their first Asterisk installation because they envision having a production-quality system in a couple of hours. This may be possible once all the concepts are learned, but few are able to do it their first time out. The intension of this book is to get you up to speed as quickly as possible, but we recommend you take your time and enjoy the process. You'll feel much happier with Asterisk if you give yourself proper time to fall in love with it (and make no mistake; fall in love you will).

Don't like it? You can change it!

Asterisk is open source software. The ability to read the source code is its power. Most other PBX systems are entirely closed-source, limiting you to the features the designers decided you could have. In Asterisk, if something does not work quite the way you want it to, you can change it! Naturally you will not be able to do that yourself without sufficient programming skills, but then again, such skills are easily hired from within the very same community that developed the product. Try THAT with a proprietary system.

The source code is also an excellent debugging and learning resource in itself. Reading through the many text and code files in the Asterisk source directory can teach you much about the intricate workings of the system.

Free and Open Source Software: GPL and LGPL Licensing

The Gnu Public License is an exciting concept, and much has been written about it. The Free Software Foundation (www.fsf.org), would probably be the best place to begin researching the GPL.

The concept of the GPL with respect to Asterisk works something like this: With most PBXs, you need to pay huge licencing fees, merely to install the software. With Asterisk, all you need to pay is attention to the obligations the GPL places on you. Stated simply, if you are not willing to abide by the terms of the GPL, then you will have to either a) cease using Asterisk, or b) negotiate a separate licencing agreement with Digium, the holders of the copyright on Asterisk.

Feel free to download and use Asterisk as you want; the GPL is all about freedom. But be aware that by using any GPL software, you are agreeing to be bound by the terms of the GPL.

Chapter 2. Preparing Your System for Asterisk

This chapter will help you to prepare you system for the installation of Asterisk. Asterisk will work on many platforms and operating systems, but we have chosen to keep things as simple as possible for you and stick to a single platform and Linux distribution. These instructions may work for you on a different Linux distro, but they have not been tested.

Hardware

Platform and Digium Hardware

While Asterisk has been successfully compiled and run on platforms other than x86, it is currently only supported on that platform. To put it simply, the ideal system for a devlopment, hobby or educational system is any PC that you can install Linux on. For more intensive and demanding environments, the selection of a platform requires a thorough understanding of system architecture, load balancing, and such. That discussion is beyond the scope of this document.

Hardware Minimums

Asterisk can be very processor-intensive, due to its use of the CPU to perform Digital Signal Prosessing (DSP). If you are building a complex, high-usage system, this is extremely important to understand. However, for the purposes of building your first Asterisk PBX, you can safely use any Intel-compatible PC (x86) that's better than, say, a 300MHz Pentium with 256 megs of RAM. Essentially, if you can install and run Fedora Core 1 on it, it should be suitable for your first Asterisk install (the authors have run Asterisk on systems ranging from 700MHz Celerons to Athlon XP 2000s).

Asterisk doesn't require very much space (approximately 100MB compiled) but sourcecode, voicemail, custom prompts, all require storage. You should be able to easily build a development system with approximately 4 GB of hard drive space.

If you are building a VoIP-only system, no extra hardware is required to compile and use Asterisk. Your extensions can be VoIP using freely available softphones such as the Xten X-Lite SIP phone. Lines can be provided through a VoIP carrier such as FWD. A VoIP-only system will allow you to evaluate Asterisk without any cost other than the computer system. However, in order to fully explore the power of Asterisk you will find yourself wanting to install some of Digium's hardware. We recommend that you consider obtaining a PCI development kit from Digium.

> **Note:** Many people running VoIP-only Asterisk systems require a clocking source to provide timing. The Digium cards have this capability built-in, so if you have one of those cards you've got the required clock. For systems without a clocking source, there's ztdummy. The ztdummy driver uses the USB controller as a timing source for the applications which require timing, such as the MeetMe conferencing application. There are two main types of USB controller chips used on motherboards. These include UHCI and OHCI type chips. For the ztdummy to work, you will require the UHCI-type USB controller. OHCI-type chips will also work, but they require the zaprtc module. The entire ztdummy module falls outside the scope of this document, but we wanted to mention it because the lack of a timing source may affect a VoIP-only system.
>
> The ztdummy module is generally considered a hack and is not really intended for production systems. Any of the Digium cards are the preferred source for timing.

Assembling Your System

The hardware platform required by the Asterisk PBX is really not much more than a stripped down PC. You will not require a powerful video card, and peripherals such as serial ports, the parallel port, or USB ports can be completely disabled. Only a network card will be essential.

If you are using any Digium cards, you will probably want to refer to your motherboard instructions to determine which PCI slots will be suitable for those cards. Many motherboards share interrupts on the PCI slots, and interrupt conflicts are a potential source of audio quality problems in Asterisk.

One possible way to free up IRQs is to disable any devices in the BIOS which are not required. Serial ports, Parallel ports and USB ports are examples of resources that Asterisk does not require, so you might consider freeing up the IRQs in order to make resource allocation easier.

IRQ Sharing Issues

Many telephony cards such as the X100P can generate a large amount of interrupts; servicing them takes time. Drivers may not be able to do it on-time if another device is processing the same shared IRQ and the IRQ line cannot receive another one. It tends to work better on SMP (APIC) systems. On single chip systems you can get interrupt misses and misaligned clocking. Any of Digium's cards or other telephony cards can be subject to this problem. Because the precise delivery of IRQs is very much necessary in telephony, one should not share IRQs with anything. This is not to say you will necessarily have IRQ sharing conflicts, but it is something to be aware of.

If you are dedicating the computer to Asterisk, free up as many IRQs as possible by disabling USB, serial and parallel port support in the BIOS. Essentially you want to free as many IRQs as possible. You will *not* want to see a NIC sharing an IRQ with a TDM or FXO card! It's best that these cards have their own IRQ.

Most BIOS' will allow you to manually assign IRQs to specific slots. Go into the BIOS and look for the IRQ area, often on the second page. If it is set to AUTO by default, try setting to manual and see what happens. Chances are a table will become available for manually assigning an IRQ to each slot.

> **Note:** One thing to watch out for on some motherboards is slots *sharing* IRQs. Look in the BIOS and check if it has an entry like "1/5" in the IRQ table. This is not at all desireable for a system that will be running Digium Zaptel boards. Generally this kind of feature is one of the differentiators between low and high cost motherboards.

Once booted, view `/proc/interrupts` to see the assigned IRQs.

> **Note:** The output below is just an example with Digium hardware flagged. This will only be available after you have loaded the hardware drivers (which is explained in chapter 3) but you should be aware of the issue.

```
#
cat /proc/interrupts

           CPU0
  0:   41353058          XT-PIC  timer
  1:       1988          XT-PIC  keyboard
```

```
    2:          0          XT-PIC  cascade
    3:  413437739          XT-PIC  wctdm <-- TDM400
    4:    5721494          XT-PIC  eth0
    7:  413453581          XT-PIC  wcfxo <-- X100P
    8:          1          XT-PIC  rtc
    9:  413445182          XT-PIC  wcfxo <-- X100P
   12:          0          XT-PIC  PS/2 Mouse
   14:     179578          XT-PIC  ide0
   15:          3          XT-PIC  ide1
NMI:          0
ERR:          0
```

Above you can see the three Digium cards each on its own IRQ. If this is the case, you can go on to install the hardware drivers. If it is not the case, you are strongly advised to tweak your BIOS until the Digium cards are not sharing IRQs with other devices.

Installing Cards

If you have purchased an Asterisk developer kit from Digium, you will be able to connect your system directly to the Public Telephone Network, as well as run an analog phone from it. If you do not have a development kit, you will be restricted to using IP trunking and clients.

Choosing an Operating System / Distribution

The Asterisk PBX was originally designed for the Linux operating system. Due to Asterisk's increasing popularity (and an active developer community), it has since been successfully ported to BSD and OS X. Nevertheless, Digium's PSTN cards are designed to work in a Linux i386 system, and Linux is still the officially supported OS, so we recommend that people new to Asterisk use Linux.

Linux Requirements

Tested Linux Distributions

Asterisk works with most Linux distributions. Many distributions such as RedHat, Fedora, Debian, Mandrake, Slackware and Gentoo have all been used successfully by the developers. If you find something does not work on any particular system, you may want to file a bug report. See http://www.digium.com/bugtracker.html[1]

Minimal Kernel Version

Asterisk is designed to work on Linux kernel version 2.4, however there is some support for kernel 2.6. If you are trying to build a stable system, it is recommended that you use the 2.4 kernel. The following sections will assume you are running on a 2.4 kernel based system.

Required Packages

Previously there were some packages that were requirements to install Asterisk such as readline and readline-devel that are no longer required. There is no special hardware needed such as a soundcard and the only required package is Asterisk itself. If you are using Digium hardware or ztdummy, you will need the zaptel package. The zaptel package is required for some Asterisk applications to be included at compile time. If you choose to compile Asterisk and not zaptel, but find that you are missing an application related to the zaptel package (ie. MeetMe), you will have to compile Zaptel and then re-compile Asterisk for the application to be included. For T1 and E1 interfaces the libpri package is required. Bison is required to compile Asterisk. The ncurses and ncurses development packages are required if you wish to build the newt tools (e.g. astman).

> **Note:** As of October 18, 2004, the zlib and zlib-devel packages are now required to compile CVS-HEAD. This is due to the addition of the DUNDi (Distributed Universal Number Discovery) protocol. These can be installed with the use of **yum** by executing the command **yum install zlib zlib-devel** from the command line.

Fedora Linux vs. Other Distributions / OSes

For our installation the authors have chosen to use the Fedora Core distribution of Linux. A frequently asked question is "What distribution of Linux is the best to run Asterisk on?" and the answer to that is "What distribution are you most familiar with?". Asterisk will run on most (if not all) distributions of Linux, so feel free to install Asterisk on your favourite. In order to help minimize the possibility of errors encountered during the first time installation of Asterisk, we have decided to use a single distribution. Fedora Core was chosen as it was the distribution most familiar to the authors.

Obtaining FC1

Fedora Core 1 and 2 are available from http://fedora.redhat.com/[2] . The major difference between the distributions is the use of the Linux 2.4 or 2.6 kernel respectively. Our installation is going to be based on Fedora Core 1 using the 2.4 kernel - the recommended kernel to run Asterisk on. These instructions should work for either distribution, but where differences lie we will note them.

Overview of FC1 for Asterisk

Production/Clean vs. Experimental/Hobby System

If an Asterisk system is being deployed in a production environment, careful consideration needs to be paid to the design of the system. Not only should the Asterisk server be optimized to ensure the telephony functions receive the highest priority, but in most cases the Asterisk system should not be running any other processes. If database or web server functionality is desired, careful thought should be given towards deploying those services on a secondary, supporting server, as opposed to on the same platform as Asterisk

Having said that, the performance possible with modern PCs is such that you can run five (or likely more) telephones on a hobby system with no problems, even if a full install of your favorite Linux distribution was selected.

The bottom line is that Asterisk can be installed quite easily on a Linux system that has all the bells and whistles, but if you wanted to build a 30 or more station PBX, you'd better the operating system down nothing other than what Asterisk requires.

What is Required

The following software will be installed:

- Zaptel (the drivers for analog cards)
- Asterisk (the PBX)
- Bison (required to compile the source code)

Stick with us through the rest of this document, as these items will be explained in more detail.

What is Not Required

Asterisk is a very performance-sensitive application. What this means is that you must carefully consider the system resources used by any other applications you are running. The most significant and unnecessary application in this regard would be the X Windowing system, popularly represented in Linux by the Gnome and KDE desktops. Not only is this not required, but it is arguably the worst thing you can run on your Linux system if you want to run Asterisk as well. Remember that Asterisk is a server application, so a desktop is not required.

You can successfully run and install Asterisk alongside X, but please be aware that it is not a good idea from a performance standpoint, and you should never put a system so configured into a production environment. If you want to properly learn Asterisk, you'd do well to leave X out of the equation.

Latest Patches and Updates

The Fedora Core distribution of Linux contains a very handy utility called **yum** . With this utility we can make sure all our packages are updated to the latest secure, stable version. After we have installed FC1, the very first command we should run after logging in is **yum update** . This will contact the main yum mirrors to pull the latest updates and install them and their dependencies for you. We want to make sure we run this command before compiling and installing the Asterisk software. **yum** will also install the latest kernel for you, but you will have to reboot your system for it to take effect. Do this before you compile Asterisk, or Asterisk will no longer work and you will have to recompile before it will start for you again.

Installing FC1 for Asterisk

Initial Steps

Insert your first Fedora Core 1 disk and boot off of it. Once you see the splash screen, type **linux text** to boot into the non-graphical installation. You will then be asked to test the media before installing. Feel free to test the media at this point in order to possibly save some installation problems.

Fedora will then probe for your hardware. After this completes you will be shown the Fedora Core welcome screen. Press *OK* to continue.

The following screens will allow you to customize your language, keyboard and mouse model. Make the appropriate selections for language and hardware.

You will then be prompted for the type of system you would like to install. Highlight *Custom* and select *OK* .

Disk Partitioning

We must now partition our hard disk drive. Fedora gives us the options of either letting Fedora create the partitions for us - *Autopartition* - or allowing us to create the partitions ourself using *Disk Druid* . If you are comfortable with partitioning the hard drive yourself or would like to customize it, select *Disk Druid* . We will assume you will be selecting *Autopartition* .

> **Warning**
>
> It is being assumed that you have built a system which will be used exclusively for Asterisk. If you have extra hard drives or partitions with information you wish to keep, be very careful about *not* destroying this information.

Select *Remove all partitions on this system* and select *OK* . You will then be shown the status of the partitions which Fedora will create. If these are fine, select *OK* .

Boot Loader

You will be given the options of *Use GRUB Boot Loader* or *No Boot Loader* . Select the default *Use GRUB Boot Loader* and select *OK* .

Some systems need to pass special information to the kernel at boot time. If you don't need any or aren't sure just leave it as the default and select *OK* .

If you would like a password on your boot loader insert it now or select *OK* to continue.

> **Note:** You would normally not want a boot loader password on a production system, as you would effectively prevent the system from recovering from a reboot without human intervention. On a lab system, a boot loader might be an effective way to prevent people from messing with your new toy.

The next screen shows the boot label and the device it will boot off of. The defaults should be fine - select *OK* to continue to the next screen where you will be asked where you want the boot loader installed to. For most installations you will want the default of *Master Boot Record* . Select *OK* to continue.

Network Configuration

If you have network cards installed in the computer Fedora will ask you to configure them. Configure the network cards for your network and select *OK* to continue to the firewall configuration.

Using a firewall is always a good idea whenever exposing a computer to the Internet. Asterisk will work fine behind a firewall as long as the appropriate ports are opened. For example, SIP uses port 5060 for the communication messages but dynamically uses other ports for the RTP audio stream. The RTP ports must be open in order to allow the audio through the firewall, and thus be able to hear the calling party. IAX2, which uses port 4569, has fewer problems with firewalls since its communication message and RTP audio streams all run across a single port number. As this is our first installation of Asterisk, and we would like to limit the possible number of system configuration issues, we are going to disable the firewall. However, be aware that you

will need to properly secure your Asterisk PBX before deploying it or attaching it to the Internet. Securing your Asterisk system is outside the scope of this document.

Language Support and Time Zone Selection

Select *English (USA)* as the language and select *OK* to continue. You will then be asked if your system clock is set to UTC time and what time zone you are in. Make the appropriate selections and select *OK* .

Setting the Root Password

Set and confirm your root password for the system.

Package Selection

Deselect *all* packages including the X Window system. Then select the following 3 packages from the list:

- Editors
- Development Tools
- Kernel Development

> **Note:** If you are using Fedora Core 2, you will not need to install the the Kernel Development package as you will be setting up the symlink required to point to `/lib/modules/‘uname -r‘/build` . This will be explained later.

While selecting other packages may not necessarily give you problems compiling or running Asterisk, we are sticking to our minimalistic approach by only selecting the packages we require to obtain, compile, install and configure Asterisk. The above three packages will allow us to accomplish this goal. Select *OK* to continue.

The next screen will tell you that installation is ready to begin. Select *OK* to continue the package installation. Fedora will ask you to confirm that you have all three installation CDs. Select *continue* to start copying and installing the operating system.

Post Install

If you would like to create a boot diskette, do so now. If not, select *No* to continue. Your system should now be installed. Select *Reboot* to reboot the system and start Fedora Core 1. After you reboot, we will run a few commands to complete our system preparation for Asterisk.

Once your system has booted successfully, login as root. After you have logged in, run the **yum update** command to update the system files and kernel. Fedora will download the RPM header files and resolve dependencies. After this completes the system will ask you if it is ok to install these files. Press **Y** for *yes* and hit **enter** . Once the system files have been updated, reboot your system for the new kernel to take effect.

After the system reboot, verify the symlink to the kernel source. Perform the following commands:

```
cd /usr/src/
```

```
ls -la
```

The symlink `linux-2.4` should be pointed to your kernel sources. You can verify that the symlink is for your current kernel and not the old kernel by comparing the version number shown in the symlink against the output of **uname -r** .

```
#
 ls -la
  lrwxrwxrwx   1 root root   24 Sep  5 12:36 linux-2.4 -> linux-2.4.22-1.2199.nptl


#
 uname -r
  2.4.22-1.2199.nptl
```

The above example shows that the symlink is correct for the currently running kernel. If the link is not correct, we can create the symlink manually with the **ln** command.

```
#
 ln -s linux-2.4.22-1.2199-nptl linux-2.4
```

> **Note:** If running Fedora Core 2, your symlink will point to a different place. The 2.6 kernel includes a directory which you can build your software against instead of the kernel source code. The symlink can be created with the command: **ln -s /lib/modules/‘uname -r‘/build** .

Your Linux system should now be ready to install Asterisk.

Conclusion

You should now have a fully installed Fedora Core 1 (or 2) Linux distribution ready for Asterisk. The next chapter will deal with obtaining, compiling and installing Asterisk.

Notes

1. http://www.digium.com/bugtracker.html
2. http://fedora.redhat.com/

Chapter 3. Obtaining and Compiling Asterisk

Getting Asterisk from CVS

What is CVS?

CVS is a central repository which developers use to control the source code. When a change is made it is committed to the CVS server where it is immediately available for download and compilation. Another added benefit to using CVS is that if something was working at one point, but a change causes it to break, the version for any particular file can be rolled back to a certain point. This is true for the entire tree as well. If you find something was working at one point but installing the latest version of Asterisk causes that to break, you can "roll-back" to any point in time. See the section on getting the files from CVS.

Telephony Card Drivers

Obtain

To obtain the Zaptel drivers for use with Digium hardware, you will have to checkout the zaptel branch from the Digium CVS server.

Example 3-1. Checkout Zaptel Drivers from CVS

```
cd /usr/src/

export CVSROOT=:pserver:anoncvs@cvs.digium.com:/usr/cvsroot

cvs login

password is anoncvs

cvs checkout zaptel
```

You will be connected to the CVS server where it will download all the files necessary to compile the Zaptel drivers. These files will be stored in the `/usr/src/zaptel/` directory.

Compile

Zaptel

You will need to compile the Zaptel modules if you plan on using ztdummy or any Digium hardware. The following commands will compile and install the modules for any Digium hardware which you might have installed in your system. See the section regarding compiling ztdummy if you do not have any Digium hardware.

Example 3-2. Compiling Zaptel Drivers

```
cd /usr/src/zaptel/

make clean

make install
```

Note: If using *Fedora Core 2* or any distribution which uses the 2.6 kernel, you will need to do an additional step before the **make install** . If using a 2.6 kernel, perform the following commands:

```
cd /usr/src/zaptel

make clean

make linux26

make install
```

Compiling ztdummy

Verify the USB Modules

The ztdummy module is used when you do not have any Digium hardware for timing but need it - such as for Music On Hold or Conferencing. The ztdummy driver requires that you have a UHCI USB controller chip on your motherboard. If you are using an OHCI USB controller you will need to use zaprtc. You can check to see if your motherboard has the UHCI USB controller by running **lsmod** from the command line.

```
#
lsmod
 Module                  Size  Used by    Not tainted
 ...
 usb-uhci               24524   0  [unused]
<-- usb-uhci
 usbcore                71680   1  [hid usb-uhci]
 ...
```

The above screenshot shows the loaded USB modules. You are looking to see the line which reads *usb-uhci* . This shows that the UHCI module is loaded and ready to be used by the ztdummy module.

Editing the Makefile

To compile the ztdummy module you will have to edit the `Makefile` located in your `/usr/src/zaptel` directory. Find the line containing -

```
MODULES=zaptel tor2 torisa wcusb wcfxo wcfxs \
        ztdynamic ztd-eth wct1xxp wct4xxp # ztdummy
```

- and uncomment the ztdummy module by removing the hash mark (#). Save the file and perform the compilation as normal. Once you have successfully compiled the ztdummy module you can load it into memory by using **modprobe** .

Modprobe

Modprobe is used to load the zaptel drivers into memory so that we can access the hardware in our system. We always load the zaptel driver into memory first. After that, we then load the driver specific to the type of device we are loading (FXS, FXO, ztdummy, etc...)

Zaptel

We can load the Zaptel module with the following command.

Example 3-3. Loading the Zaptel Driver

```
modprobe zaptel
```

If the zaptel module loaded successfully, you shouldn't see any output after hitting enter. We can verify the zaptel module loaded successfully by running the **lsmod** command. We should see something similar to the following if zaptel loaded.

```
#
lsmod
Module                  Size  Used by    Not tainted
zaptel                178144   0  (unused)
<-- zaptel

 ...
```

As we can see, the very first module listed is zaptel. The zaptel module is used by our channel type modules, thus why it is currently unused. This will change once we load the modules for the FXO and FXS ports.

We can also check in /var/log/messages using the **tail** command. We should see a line which says the Zapata telephony interface was registered.

```
#

tail /var/log/messages

 Sep  5 13:44:47 localhost kernel: Zapata Telephony Interface Registered
 on major 196
```

ztdummy

If we don't have any Digium hardware, we will be required to use the ztdummy module. Remember that we need to have a UHCI USB controller as the timing source for the ztdummy module. We load the ztdummy module very similar to the zaptel module using the **modprobe** command. You load the ztdummy module *after* the zaptel module since ztdummy is dependent on zaptel.

Example 3-4. Loading the ztdummy Module

```
modprobe ztdummy
```

Again, we should not see any output from the modprobe command. We can verify our ztdummy module loaded correctly with the use of **lsmod** .

```
#

lsmod

 Module                  Size  Used by    Not tainted
 ztdummy                 2580   0  (unused)
<-- ztdummy

 zaptel                178144   0  [ztdummy
<-- zaptel

 ...
 usb-uhci               24524   0  [ztdummy]
<-- usb-uhci used by ztdummy

 usbcore                71680   1  [hid usb-uhci]
```

Looking at the output of **lsmod** we can see that our ztdummy module is loaded correctly. Notice that both the zaptel and usb-uhci modules are being used by ztdummy. This makes sense since zdummy relies on both zaptel and usb-uhci.

Configuring Zaptel parameters

In order to configure regional parameters and signalling for physical telephony channels, the zaptel.conf file needs to be edited. This file contains many options and parameters, which will not be included in this book

For the purposes of this document, we will limit ourselves to what would be included in a dev kit, which is a single FXO interface, and a single FXS.

Editing `/etc/zaptel.conf`

The `zaptel.conf` file is located in the `/etc/` directory

```
#
 cd /etc/
```

Open the `zaptel.conf` with your favourite editor. You will notice a large number of commented lines. You can read this information for your own knowledge, but for now we will ignore most of it. The only two lines that are uncommented by default come near the end of the file; the loadzone and defaultzone parameters. For now leave these as the default.

> **Note:** The loadzone parameter allows you to specify the tones that can be generated by that interface. A comma separated list of two letter country codes will load the tones for that country or region. The currently available list of tone zones defined in the `zonedata.c` file in `/usr/src/zaptel` .

Within the `zaptel.conf` file, we define the type of signalling that the channel is going to use. We also define which channels will be loaded. This is the information that will be used to configure the channels with the use of the **ztcfg** command which will be discussed later in this chapter. When you are dealing with an FX interface, the hardware is described based on what it connects to, (ie. an FXO interface is thus named because it connects to an *Office* , where as an FXS interface, connects to a *Station*); the signalling however, needs define the device we are emulating - so, since an FXO interface is connecting to an Office, we need software to emulate a station. The opposite is then true for an FXS interface; we need the software to emulate an Office, thus, FXO. In summary, the hardware is described by what it conects to, but the software, or the driver, is defined by the behaviour that it needs to exhibit.

> **Note:** It is important to note that how FXO and FXS devices are described in software can vary among the various manufacturers. The FXO and FXS descriptions for hardware are universally consistant from what the authors can determine.

Since we are assuming that you will be configuring the `zaptel.conf` file for use with the TDM400P devkit which contains an FXS and FXO interface, our examples below will focus on this card. However the concepts which will be learned can be applied to other interfaces, including adding more channels through FXO and FXS modules on the TDM400P card.

Example 3-5. Configuring `zaptel.conf` for use with a TDM11B

```
#
# Zaptel Configuration File
#
fxoks=1
fxsks=4
loadzone=us
defaultzone=us
```

The TDM11B (a single FXS and single FXO module on a TDM400P card) comes standard with the FXS module attached to the first port of the TDM400P. The FXO module is then attached to the fourth port of the card. The **fxoks=1** line then tells the wcfxs module to use FXO signalling on the first port of the TDM400P. Likewise, the **fxsks=4** line tells the module to load the fourth port of the card with FXS signalling. The **ks** part of the line specifies to use the koolstart supervision protocol. The koolstart protocol is not an industry known supervision mechanism. What koolstart does is add intelligence to the circuits ability to monitor what the far end is doing. This is a cool feature, thus it receives the name koolstart. Since koolstart incorporates the advantages of loopstart and groundstart (ls and gs respectively) and is superior to both, you'll almost always want to use it unless you have a compatibility problem. Koolstart has become the informal standard for use with Asterisk.

Loading the Driver

After configuring the `zaptel.conf` configuration file, you must load the wcfxs driver using **modprobe** . This is very similar to how we loaded both zaptel and ztdummy earlier in this chapter.

> **Note:** If you loaded the ztdummy module earlier in this chapter, it is recommended that you unload the driver before loading the wcfxs module. You can do this with the use of the **rmmod** command. To unload the ztdummy module, perform the following command:
>
> **rmmod ztdummy**
>
> If successful, you shouldn't see any output.

The wcfxs driver can be loaded with the use of **modprobe** command as follows:

```
#
 modprobe wcfxs

   Freshmaker version: 71
   Freshmaker passed register test
   Module 0: Installed -- AUTO FXS/DPO
   Module 1: Not Installed
   Module 2: Not Installed
   Module 3: Installed -- AUTO FXO (FCC mode)
   Found a Wildcard TDM: Wildcard TDM400P REV E/F (4 modules)
```

As can be observed, the module performs a series of tests and outputs the results to the screen. This allows you to see that the kernel module loaded the information about the physical interface correctly. After loading the driver, you must configure the channels with the use of **ztcfg** . This is explained in the following section.

ztcfg

The **ztcfg** command is used to configure the signalling used for the physical FX interface. **ztcfg** will use the signalling configuration specified in `zaptel.conf` . In order to see the output and result of the **ztcfg** command, we must append a -vv to make the program verbose.

```
#
```

```
/sbin/ztcfg -vv
```

```
Zaptel Configuration
======================

Channel map:

Channel 01: FXO Kewlstart (Default) (Slaves: 01)
Channel 04: FXS Kewlstart (Default) (Slaves: 04)

2 channels configured.
```

If the channels loaded successfully, you will see an output similar to the above. A common mistake is that the signalling of the channels gets reversed in the `zaptel.conf` file. If this happens, you will see the following output:

```
ZT_CHANCONFIG failed on channel 1: Invalid argument (22)
Did you forget that FXS interfaces are configured with FXO signalling
and that FXO interfaces use FXS signalling?
```

After the channels are successfully configured, you are ready to start using your hardware with Asterisk.

zttool

Asterisk

Obtain

In order to obtain Asterisk, you must check it out from the Digium CVS server. Currently there are two flavours of Asterisk which you can obtain: Release Candidate 2 (RC2) and HEAD (the development branch). The HEAD branch is the most recent version which contains all the newest patches implemented into the source. Although uncommon, like all development branches, it is possible to get a broken version of Asterisk from the CVS server. Your safest bet is to use the RC2 version which can be considered the stable version.

Obtaining the RC2 Branch

```
cd /usr/src/
```

```
export CVSROOT=:pserver:anoncvs@cvs.digium.com:/usr/cvsroot
```

```
cvs login
```

```
password is anoncvs
```

```
cvs checkout -r v1_0_rc_2 asterisk
```

Obtaining the HEAD Branch

```
cd /usr/src/

export CVSROOT=:pserver:anoncvs@cvs.digium.com:/usr/cvsroot

cvs login

password is anoncvs

cvs checkout asterisk
```

Compile

If you have compiled software before, compiling Asterisk will seem very straight forward. Run the following commands to compile and install the Asterisk software after you have obtained it from the CVS server.

Example 3-6. Compiling Asterisk

```
cd /usr/src/asterisk/

make clean ; make install
```

Test

Before you can start Asterisk, you must create configuration files. While there are many configuration files which are specific for the different applications and channels which Asterisk supports, there are very few which are required in order to start the system. We will lead you through the installation of this minimal set of configuration files in order for us to test that Asterisk will load successfully.

We must first make the directory where all the configuration files for Asterisk will reside. By default, Asterisk looks for its configuration files in the `/etc/asterisk/` directory. Lets make this directory now.

```
mkdir /etc/asterisk/
```

Now that we have a location for our configuration files, we need to create them. Luckily for us, Asterisk comes with many sample configuration files. These sample configuration files are very useful for looking up information and about a command and how it is used within its respective configuration file. These sample files are located in the `/usr/src/asterisk/configs/` directory. Lets create these required files now.

Example 3-7. Creating the Configuration Files

```
cd /usr/src/asterisk/configs/
cp ./modem.conf.sample /etc/asterisk/modem.conf
cp ./modules.conf.sample /etc/asterisk/modules.conf
cp ./phone.conf.sample /etc/asterisk/phone.conf
cp ./voicemail.conf.sample /etc/asterisk/voicemail.conf
cp ./zapata.conf.sample /etc/asterisk/zapata.conf
```

With this minimal configuration set, we should be able to start Asterisk successfully. We can do this with the following command.

Example 3-8. Starting Asterisk

```
/usr/sbin/asterisk -cvvv
```

Asterisk will start now and you should see a **CLI*>** prompt. You will notice several errors as Asterisk starts saying it is unable to find various file names. These files are currently outside the scope of this document and will be discussed in future volumes. At this point Asterisk is not fully functional, but we do know that it will start successfully. All that remains is the configuration of the system.

Chapter 4. Configuring Channels

Introduction to Channels

Channels are logical connections to the various signalling and transmission paths which Asterisk can use to create and connect calls. They can be physical (such as an analog FXO port), or software-based (such as an IAX channel), depending on their nature. The dialplan (Chapter 5) is where you define the rules which Asterisk follows to determine how you want it to connect channels together. Obviously, before we can build a dial plan, we must determine what kind of channels we need, and configure them to be useable by the system.

Channels come in all kinds of different formats; physical telecommunications circuits (such as FXO, FXS, PRI, BRI), software-based, network attached entities (such as SIP and IAX), and internal channels exclusive to Asterisk for all kinds of interesting wizardry (such as Agent, Console, Local and the very brilliant *TDMoE*). Asterisk treats all of these channels as connection points which you bring together in the dialplan. It is important to remember that even though channels may vary in terms of technology and connectivity, Asterisk allows you to treat them all pretty much the same.

Asterisk is an extremely flexible and powerful PBX in large part because of the way it handles channels. In many older, proprietary PBXs, different channels have completely different ways of communicating. Station ports, Trunk ports and IP ports are so different that it requires years of experience to figure out how to creatively connect them together. Indeed, it could be argued that becoming an expert on a particular PBX is mostly becoming intimately aware of it's limitations, and learning creative kludges to get around those limitations. Asterisk changes all that.

Wherever possible, Asterisk allows you to treat any channel type in the same manner as any other channel type. Whether you are connnected to an analog set on an FXS port, an IAX channel to IAXTel, or a SIP softphone, the functions available to you are usually going to be the same. This puts a greater responsibility on you the designer, as Asterisk will allow you to create something that is not workable. But it also means that Asterisk is flexible in ways no other PBX can be.

In this document, we are going to walk you through the creation of four different channel types: FXO, FXS, IAX and SIP. These were chosen because they are the most popular channel types currently in use. If you do not have a development kit from Digium, you may not have access to FXO and FXS cards, if that is the case, please do read the FXO and FXS section, but do not create the `/etc/asterisk/zapata.conf` file.

FXO and FXS

The terms "FXO" and "FXS" have their origins in an old telephone service called Foreign eXchange (FX). The original purpose of an FX circuit was to allow an analog phone at a remote location to be connected to a PBX somewhere else. An FX circuit has two ends (the Station end, where the telephone is, and the Office end, where the PBX is). The confusing part about understanding FXO and FXS is that FX cards are not named by what they are, but rather by what is connected to them. An FXS card, therefore, is a card that you connect a Station to. Since that is so, you can see that in order to do it's job an FXS card must behave like a central office. Similarly, an FXO card connects to a Central Office, which means it will need to behave like a telephone (a modem is a classic example of an FXO device).

FXS

Foreign eXchange Station (FXS) channels provide the same interface as the traditional analog line your phone company provides to most homes or small businesses.

Among other things, FXS channels would normally provide:

- Dial tone
- Ringing voltage
- DTMF (touch tone) detection
- Message waiting
- Calling Line ID

In Asterisk, when you configure a Digium FXS card (such as the TDM400P with FXS cards installed), you will need to define parameters that are relevant to an FXS card. *Most importantly, you need to remember that an FXS card provides the signalling of a central office* . You will want to configure `/etc/asterisk/zapata.conf` as follows:

```
language=en
context=default
switchtype=national
signalling=fxo_ks
channel => 1
```

Note: Notice the line where we define `signalling=fxo_ks` . This describes the function the card is providing. Since an FXS card performs the functions of a Central office (providing dial tone, ring voltage, message waiting, etc), we specify that the signalling we want to provide is that of a central *O* ffice - thus FXO_KS is the signaling. It's an FX *S* card, because you connect *S* tations to it, but it needs to provide the functions of a central *O* ffice, so the signalling has to be FX *O* .

The above example does not contain all of the possible options but will set up a simple FXS channel for you, which is all we are going to do for now.

Don't forget that an FXS card has to provide central office functionality to the device(s) connected to it.

FXO

An FXO card is a card that connects to a central office. A modem is a classic example of an FXO card (in fact, if you have one of Digium's old FXO cards, the X100P, it is in actual fact a modem). An FXO device must be able to:

- Generate DTMF (touch tones)
- Detect dial tone
- Detect ringing
- Detect message waiting
- Interpret Caller ID
- Signal On Hook/Off Hook condition to the far end, as well as Flash

The main difference in the settings that are provided above when configuring an FXO channel instead of an FXS channel is signalling.

In your `/etc/asterisk/zapata.conf` file we will want to add the following:

```
signalling=fxs_ks
channel => 4
```

So your the whole file should now look like this:

```
language=en
context=default
switchtype=national
signalling=fxo_ks
channel => 1
signalling=fxs_ks
channel => 4
```

In the first example we listed channel 1 as an FXS channel. To create an FXO channel on the same TDM400P card, we list all the settings for the channel and then define the channel number. Instead of only having signalling be fxo_ks though we want the signalling to be fxs_ks. Because the other settings haven't been changed (signalling=fxs_ks replaces the previous value of signalling), they stay the same. Which means that while channel 1 takes on the values language=en, context=default, switchtype=national, and signalling=fxo_ks; channel 4 takes on the values language=en, context=default, switchtype=national, and signalling=fxs_ks.

We now having a working example of a zapata.conf with an FXS channel (1) and an FXO channel (4) that we can use. Channel 4 can be connected to an analog circuit such as might be provided by your phone company. You can plug an analog telephone directly into Channel 1.

IAX

The *Inter-Asterisk eXchange* (*IAX*) protocol is an IP-based media transport protocol. To create a conversation between to people via IP you use IAX with as the method of transporting the audio. To create an IAX channel you need to set up the IAX communications in your iax.conf file.

> **Note:** IAX is pronounced "eeks" by the Asterisk developer community, although no one minds if you say "AYE-AY-EX".

General Settings

First we need to set up our IAX globally used settings:

```
[general]
port=4569         ; What port to bind to (4569 is the default for IAX2)
bindaddr=0.0.0.0 ; Which IP address on your server to bind to (if
     ; you have multiple NICs on your sever, you can
     ; restrict IAX2 to only one of them. 0.0.0.0 will
     ; allow it to work on all NICs in your system.
deny=all            ; You want to disallow the use of all audio
     ; codecs to ensure that
     ; your system won't tell the far end that it can
     ; support just any codec. Then, you specifically ALLOW
     ; the codecs that your system supports.
allow=ulaw   ; The North American standard companding for G.711
allow=alaw   ; The rest of the world's companding standard for G.711
allow=gsm   ; A compressed codec, popular with Asterisk
```

The example above is very minimal and only sets up the basic settings to listen for connections and create them. The same way that Apache listens for http requests on port 80, Asterisk will listen for IAX requests on port 4569.

Defining IAX Channels

Now that we've defined the global parameters for our IAX interface to the outside world, we can create our IAX channels. IAX channels are very flexible, and are successfully used to connect to many kinds of endpoints. Digium makes an IAX-based device nicknamed the IAXy, which provides an FXS interface to support an analog telephone at the end of an IAX channel. IAX is also (naturally) the protocol used by the IAXTel network, which is what our examples will be connecting you to.

Although IAX is not an RFC standard protocol, it is enormously well respected. Many pundits predict that IAX will supplant SIP.

Note: Before creating this file you will probably want to get an IAXTel account set up.

```
      [general]
      port=4569              ; What port to use
      bindaddr=0.0.0.0    ; What IP address to bind to
      allow=all              ; Allow the use of all audio codecs
register => username:secret@iaxtel.com  ;replace username:secret with your credential

[iaxtel-out]
type=peer    ; Allow connections out
username=username   ; TYour IAXTel username
secret=password    ; Your secret password
deny=0.0.0.0/0.0.0.0   ; Not just anyone can be IAXTel
permit=216.207.245.47/255.255.255.255 ; This is a server at IAXTel
permit=69.73.19.178   ; This is a server at IAXTel

[iaxtel-in]
type=user    ; Allow connections to come in
context=default    ; Route calls to this context
                                       ; in the dialplan
username=username   ; The IAXTel username
secret=password    ; The secret password
deny=0.0.0.0/0.0.0.0   ; Not just anyone can be IAXTel
permit=216.207.245.47/255.255.255.255 ; This is a server at IAXTel
permit=69.73.19.178   ; This is a server at IAXTel
```

In the example above you will notice that we have 2 entries to communicate with the IAXTel service. IAXTel is a free VoIP calling service and is used as a testbed for Asterisk and IAX as well, as a common communication system.

The first change you will notice actually comes in the general section. It is a line to tell IAXTel that we are here and that calls to that IAX user should be routed to your asterisk server. It's like connecting to your IM (AOL Instant Messenger, Yahoo, MSN, etc) so that when other people send you a message you get it wherever you are logged in.

We have 2 different kinds of connections to IAXTel, the peer and the user. This allows us to decide that inbound calls can come from one server and outbound calls could come from another. This is extremely useful when you are handling a major network of Asterisk servers and are using IAX for trunking the servers together.

SIP

The Session Initiation Protocol (SIP) is rapidly becoming the most widely supported VoIP protocol. Like IAX, SIP is pretty easy to set up. There are some gotcha's with the protocol though. Be aware that while your channels may be set up correctly, SIP does not handle NAT very well, and this can be a source of significant headaches. To configure SIP, you will need to create a sip.conf file

General Settings

The first thing tht needs to be done is setup the general settings. Much like IAX this allows you to make settings that all sip connections will use.

```
[general]
port = 5060                     ; Port to bind to
bindaddr = 10.78.1.90           ; Address to bind to
context = default               ; Default for incoming calls
srvlookup=yes                   ; Enable SRV lookups on outbound calls
dtmfmode=inband
allow=all                       ; Allow all codecs
```

As you can see, the settings are very similar to IAX. We have a port, address, context, and allow. The srvlookup setting is a way to look up host names. If set to yes, DNS lookups will happen on SRV records instead of A records to accomodate for load balancing. The dtmfmode setting is used to determine how Asterisk should listen for tones, such as someone dialing an extension.

Defining SIP Channels

For SIP channels to be used, clients have to be given permission to authenticate to Asterisk via SIP. Also for Asterisk to be used as a client (for something like Free World Dialup phone service via SIP, the client settings must be setup.

```
[general]
port = 5060                     ; Port to bind to
bindaddr = 10.78.1.90           ; Address to bind to
context = default               ; Default for incoming calls
srvlookup=yes                   ; Enable SRV lookups on outbound calls
dtmfmode=inband
allow=all                       ; Allow all codecs
register => FWDNumber:secretpassword@fwd.pulver.com/EXTEN

[fwd.pulver.com]
type=user
username=FWDNumber
secret=secretpassword
host=fwd.pulver.com
nat=yes
canreinvite=no

[fwd.pulver.com]
type=peer
host=fwd.pulver.com
context=default
nat=yes
canreinvite=no
```

FWD stands for Free World Dialup, a free VoIP to VoIP service that can be found at http://www.freeworlddialup.com. Communications between FWD and IAXTel are allowed and there are instructions on their respective websites on how to do this inter-service communications.

Again, you can see that we've put a register line in the general section to let the service provider know that we are the correct client for calls that would be routed to FWDNumber. You'll also notice that there is a new context in the example. The fwd.pulver.com context allows calls to FWD and from FWD to be handled by Asterisk.

The first entry is the information necessary for outbound calls to use the FWD service as a client. It is almost identical to the section under IAX for "type=user". The second entry is for authenticating inbound calls, to ensure that we're not getting fake call routing from another source.

Other Channel Protocols

Asterisk is the most standards-compliant PBX in existence. It's support of every important telephony standard is unmatched in the industry.

H.323

ISDN

MGCP

SCCP (Skinny)

VoFR

Bluetooth

Chapter 5. Brief Introduction to Dialplans

The dialplan is the heart of an Asterisk system, as it defines how Asterisk should handle calls. It consists of a list of instructions or steps that Asterisk should follow, which are triggered by digits (or characters) received from a channel or application. To sucessfully set up your own Asterisk system, you will need to understand the dialplan.

This chapter will explain how dialplans work in a step-by-step manner, and give you the skills to create your own. The examples in this chapter have been designed to build upon one another, so feel free to go back and reread a section if something doesn't make sense. While this chapter is by no means an exhaustive survey of all the possible things that dialplans can do, our aim is to cover all the fundamentals.

Introduction to Creating Dialplans

The majority of the dialplan is specified in the file `/etc/asterisk/extensions.conf`.

> **Note:** The location of your `extensions.conf` may vary, depending on how you installed Asterisk.

This file is made up of four main parts: *contexts*, *extensions*, *priorities*, and *applications*. In this section, we'll cover each of these parts, and explain how they work together to create a dialplan. When we're done, you will have created a basic, yet functional dialplan.

If you installed the Asterisk sample files, you probably have an existing `extensions.conf` file. We are not going to modify that file. Instead, we suggest you build your `extensions.conf` file from scratch. This will greatly assist your learning, and give you a better understanding of how each part of the file plays a role in the dialplan.

The sample `extensions.conf` remains a fantastic resource to learn many of the variables and assorted subtleties of dialplans. We suggest you rename it to something like `extensions.conf.sample`. You will then always have it to refer to later. Try something like this:

```
# mv /etc/asterisk/extensions.conf /etc/asterisk/extensions.conf.sample
```

Contexts

Contexts play an organizational role within the dialplan. Contexts also define scope. You can think of contexts as a way to keep different parts of the dialplan separate. As a simple example, contexts can be used to create an auto attendant menu system, where menu choice "1" in the [SalesDepartment] context does something completely different from menu choice "1" in the [ServiceDepartment] context. You can put different users in different contexts, so that when someone from [ABCWidgetsInc] dials "0" from their set they go to their own reception but when the president of [KeepIt-SimpleStoopid] dials "0" he goes directly to the PSTN operator. This could be handy if we wanted to provide different reception destinations for different companies, all sharing the same Asterisk server. Any call that Asterisk handles will begin in a certain context. The instructions defined in this context will determine what things may happen to the call.

> **Note:** Contexts are often used to create voice menus that give callers a list of extensions to choose from by pressing keys on a touch-tone phone. This functionality is commonly referred to as an auto-attendant. We'll cover auto-attendants in more depth later in the chapter.

Contexts are denoted by their name inside of square brackets. For example, if we were to create a context for incoming calls, we might define it like this:

```
[incoming]
```

All of the instructions placed after a context definition will be part of that context. To start the next context, simply define the next one.

At the very beginning of the `extensions.conf` file, there is a special context called [globals]. The globals context is where settings and variables can be defined that can be used throughout your dialplan. We aren't going to use the full power of the [globals] context, but you should be aware of why it exists.

Extensions

Within each context, we will define one or more extensions. In Asterisk, an extension is a character string which will trigger an event. A simple example would look like this:

```
exten => 555,1,Dial(Zap/1,20)
exten => 555,2,Voicemail(u555)
```

The " exten => " tells the dial plan that the next thing it sees will be a command.

The " 555 " is the actual digits received (i.e. what the caller dialed, or the "extension" number dialed).

The " 1 " or " 2 " represent the priority, which determines which order the commands for that extension will take. In our example, " 555 " will ring Zaptel channel 1 for 20 seconds, and then connect to voicemail box 555, with an 'u'navailable message.

Extensions determine how the call flows. 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 more than that in Asterisk. In our `extensions.conf` file, an extension is declared by the word "exten", followed by an arrow formed by the equal sign and the greater-than sign like this:

```
exten =>
```

This is followed by the number (or name) of the extension, a comma, the priority, another comma, and finally the application we'd like to call on the channel. We'll explain priorities and applications next, but first let's finish covering the syntax. The syntax looks like this:

```
    [
context-name
]
```

```
    exten =>
extension
,
priority
,
application
```

Priorities

Priorities are numbered steps in the execution of each extension. Each priority calls one specific application. Typically these priority numbers simply start at 1 and increment consecutively for each line in the context. Priority numbers aren't *always* consecutive, but we'll worry about that later. For now, just remember that for each extension, Asterisk runs each priority in order.

Applications

Applications perform certain actions on a voice channel, such as playing sounds, accepting touch-tone input, or hanging up the call. Options called arguments can be passed to applications to affect how they perform their actions. (To see a listing of the possible applications that are built into Asterisk, you can type **show applications** at the Asterisk command-line interface.) As we build our first dialplan below, you'll learn to use applications to your advantage.

A Simple Example

Now we're ready to create our first `extensions.conf` file. Because this is our first step, we'll start with a very simple example. We'll assume for this example that all Asterisk needs to do is to answer the call, play a sound file that says "Goodbye", and then hang up. We'll use this simple example to point out the fundamentals of creating a dialplan.

The special 's' extension

Before we get too far into our example, we need to cover a special extension called 's', which stands for "start". By default, calls will start in a context's 's' extension. (You can think of 's' as an extension that gets automatically executed.) In our example, we'll start by creating a dialplan with this 's' extension.

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 which is ringing. This does the initial setup for the call so that we can perform other functions. A few applications don't necessarily require that we `Answer()` the channel first, but it is a very good habit to properly `Answer()` the channel before doing anything else.

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 are usually found in `/var/lib/asterisk/sounds/`. `Playback()` is used by specifying the filename (without a file extension) as the argument. For example,

`Playback(filename)` would play the sound file called *filename* .gsm, located in the default sounds directory.

The `Hangup()` application does exactly as it's name implies; it hangs up an active channel. You would use this at the end of a context once you want to drop a caller who no longer needs to be connected to the system.

Our first dialplan

Now we're ready to start our first example dialplan. Please pay attention to the way that each priority calls an application. Note that in this example, we only have one extension. In later examples we'll add other extensions, and show how to move from one extension to another.

> **Note:** The following examples are not meant to be completely working and usable. We are simply using these examples to explain how dialplans work. For these examples to work, you should have already configured some Zap channels (using a Devkit from Digium, for example), and configured those channels so that incoming calls go to the [incoming] context.

After showing our first example, we'll explain each step.

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

When a call is sent into this [incoming] context, it will first go to the 's' extension. As we learned earlier, calls usually begin in the 's' extension. We have three priorities in this context, numbered 1, 2 and 3. Each priority calls a particular application. Let's take a closer look at these three priorities.

Our first priority invokes the `Answer()` application. Asterisk then takes control of the line and sets up the call. After answering the line, Asterisk goes on to the next priority. In our second priority, we call the `Playback()` application. This will play a sound file as specified by the filename. In our example we will play the file `goodbye` . The caller will hear a voice say "goodbye". Notice that there is no filename extension. Asterisk will automatically determine the extension of the sound file. In our third and final priority line, we call the `Hangup()` application and thus end the call.

A more useful example

Now that we've gone through a simple example, let's build upon it by learning about the `Background()` and `Goto()` applications. These two applications will allow us to create dialplans with much more functionality.

The key to interactive Asterisk systems is the `Background()` application. It gives you the ability to play a recorded sound file, but when the caller presses a key it interrupts the playback and goes to the extension that corresponds with the dialed digits.

Another very useful application is called `GoTo()` . As its name implies, it jumps from the current context, extension, and priority to the specified context, extension, and priority. The `GoTo()` application makes it easy to programatically move 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 =>
extension
,
priority
,Goto(
context
,
extension
,
priority
)
```

In this example, let's assume we've been asked by a local movie theater to create an interactive system where callers can dial in and listen to pre-recorded movie listings. To make this example simple, we'll say that the movie theater only has two screens.

We'll also assume that we have three pre-recorded sound files. The first, called `current-moves.gsm`, says "Welcome to our movie theater. To hear what's playing on screen one, press one. To hear what's playing on screen two, press two." The other two sound files, named `movie1.gsm` and `movie2.gsm` respectively, tell the caller the information about the movie playing on that particular screen.

Note: We'll cover recording sound files later in this chapter.

```
[incoming]
exten => s,1,Answer()
exten => s,2,Background(current-movies)
exten => s,3,Hangup()
exten => 1,1,Playback(movie1)
exten => 1,2,Goto(incoming,s,1)
exten => 2,1,Playback(movie2)
exten => 2,2,Goto(incoming,s,1)
```

Let's go through this example step by step. When the call enters the system, Asterisk executes the 's' extension automatically, starting with priority one. You may notice that the 's' extesion looks almost identical to our first example. The difference is the use of the `Background()` application instead of `Playback()`. As we explained above, the `Background()` application allows us to accept digits from the caller while the sound file is being played. While the `current-movies.gsm` file is being played to the caller, let's say the user presses **1**. Asterisk will then look for an extension in our current context that matches it. When Asterisk finds the '1' extension, it will execute all the priorities for that extension.

Now that the user has pressed **1**, Asterisk can perform both priorities for extension 1, which correspond to lines four and five in the above example. The first priority for extension 1 will use the `Playback()` application to play the movie details for screen one. After the file finishes playing, it will execute the second priority, which is a call to the `Goto()` application.

`Goto()` allows us to send the caller anywhere in our dialplan. The format for `Goto()` is (context,extension,priority). In our example **exten => 1,2,Goto(incoming,s,1)** we will send the user back to the first priority of the 's' extension in our 'incoming' context.

If the user doesn't press a key before the `Background()` application finishes playing the file, the third priority of our 's' extension will be performed, hanging up the user. This is probably not the best way to handle incoming calls, but we'll do it for now

as we learn the fundamentals of how to control call flow. thinking about how we can move the user around our dialplan.

Calling channels with the `Dial()` application

We are going to add to our movie theatre example by showing the use of the `Dial()` application. If the caller presses **0** during playback it will ring the ticket office. We are going to assume that the channel for the ticket office has already been set up.

```
[incoming]
exten => s,1,Answer()
exten => s,2,Background(current-movies)
exten => s,3,Hangup()
exten => 1,1,Playback(movie1)
exten => 1,2,Goto(incoming,s,1)
exten => 2,1,Playback(movie2)
exten => 2,2,Goto(incoming,s,1)
exten => 0,1,Dial(Zap/1)
```

If you compare this example with the previous one, it should be quite obvious that all we've done is add another extension to our [incoming] context. We have added extension zero which will then execute the Dial application. The `Dial()` application allows us to call a specified channel using the format of [technology]/[resource]. In this instance we are calling the first channel on our Zaptel interface (identified as "Zap/1"), which is most likely connected to an analog telephone. When the caller presses **0** at the voice menu, the phone will ring. If the operator answers the call, he or she will be connected to the caller.

By this point in the chapter you should understand the use of several applications such as `Answer()`, `Playback()` , `Background()`, `Hangup()`, `GoTo()` and the basics of `Dial()` . If you are still a little unclear of how these work, please go back and read this section again. The basic understanding of these applications is essencial to fully grasp the concepts which will be explored further.

With just a basic understanding of extensions, priorities and applications it is quite simple to create a basic dialplan. In the next few sections, we'll add on to this foundation and make our dialplan even more powerful.

Adding Additional Functionality

Handling Calls Between Internal Users

In our examples thus far we have limited ourselves to a single context. It is probably fair to assume that most installations will have more than one context. One great use of contexts is to give outside callers a different experience than people calling from inside extensions. Besides providing a different user experience, contexts are also used as a mechanism to separate privileges for different classes of callers, where the privilege might be making long-distance calls, or calling certain extensions. In our next example, we'll create a simple dialplan with two internal phone extensions, and set up the ability for these two extension to call each other. To accomplish this, let's create a new context called [internal].

> **Note:** As in previous examples, we're assuming that the channels have already been configured. We're also assuming that any calls originated by these channels begin in the [internal] context.

```
[incoming]
exten => s,1,Answer()
exten => s,2,Background(current-movies)
exten => s,3,Hangup()
exten => 1,1,Playback(movie1)
exten => 1,2,Goto(incoming,s,1)
exten => 2,1,Playback(movie2)
exten => 2,2,Goto(incoming,s,1)
exten => 0,1,Dial(Zap/1)

[internal]
exten => 1001,1,Dial(SIP/1001)
exten => 1002,1,Dial(SIP/1002)
```

The above example shows that incoming calls to the [incoming] context can dial channel Zap/1 by selecting the option **0** from our menu. However we would like to standardize the internal extensions with 4 digit numbers. As we have it set up, the caller on channel SIP/1 can call channel SIP/2 by dialing 1002, and SIP/2 can call SIP/1 by dialing extension 1001.

Variables

Variables are very useful tools in a dialplan. They can be used to help reduce typing, add clarity, or add additional logic to a dialplan. Think of a variable as a container. When we make reference to a variable, what we really want is the value that the variable contains. For example, the variable ${JOHN} might be assigned the name of the channel assigned to someone named John. In our dialplan, we could refer to the channel as ${JOHN}, and Asterisk will automatically replace it with whatever value has been assigned to the variable. References to variables are denoted by a dollar sign, an opening curly brace, the name of the variable, and a closing curly brace.

There are three types of variables; namely global variables, channel variables and environment variables. As their names imply, global variables apply to all extensions in all contexts, while channel variables only apply to the current call in progress. Environment variables are a way of accessing Unix environment variables from within Asterisk.

> **Note:** The terms arguments and variables are often used interchangably to mean the same thing. For our purposes we are going to make a firm distinction between the two. Arguments are values passed to the built-in applications to tell them what to do. Variables are containers which are assigned a particular value. They can either be used by the user or, much more frequently, by Asterisk itself.

Global Variables

Global variables are denoted using the form VARIABLE_NAME=value, and should be placed in the [globals] context. Global variables are useful as they allow us to use them within our dialplan to make it more readable. They can also greatly simplify the task of updating a dialplan. Global variables can also be defined using the `SetGlobalVar()` application.

Channel Variables

A channel variable is a variable (such as the Caller*ID number) that is associated with a particular call. Unlike global variables, channel variables are only defined for the duration of the call. There are many predefined channel variables for use within the dialplan listed in `/usr/src/asterisk/doc/README.variables`. Channel variables can also be set with the use of the `SetVar()` application.

Environment Variables

Environment variables allow us to access Unix environment variables from within Asterisk. These are referenced in the form of ${ENV(foo)} where (foo) is the environment variable you wish to reference.

Adding Variables

Lets expand upon our movie theatre example by assigning some channel names to global variables.

```
[globals]
RECEPTIONIST=Zap/1
JOHN=SIP/1001
MARY=SIP/1002

[incoming]
exten => s,1,Answer()
exten => s,2,Background(current-movies)
exten => s,3,Hangup()
exten => 1,1,Playback(movie1)
exten => 1,2,Goto(incoming,s,1)
exten => 2,1,Playback(movie2)
exten => 2,2,Goto(incoming,s,1)
exten => 0,1,Dial(${RECEPTIONIST})

[internal]
exten => 1001,1,Dial(${JOHN})
exten => 1002,1,Dial(${MARY})
```

Instead of directly assigning the channel name as the argument in our application, we can substitute a variable name. This makes it easier to update our dialplan incase we wish to the change the channel associated with our variable. Our dialplan benefits by becoming easier to read.

A Useful Debugging Tip

The `NoOp()` application (No-Operation) is useful for debugging purposes. It can be used to echo information to the Asterisk console. For example, Zap channels don't print the caller ID information on incoming calls, but we can do the following:

```
exten => s,1,Answer()
exten => s,2,NoOp(${CALLERID})
```

The CallerID information will then be output to the Asterisk console with of the predefined channel variable ${CALLERID}.

Call Flow

Pattern Matching

Extensions are not limited simply to numbers. We are able to match patterns of numbers to control our call flow. To do this we start an extension "number" with an underscore symbol (_). Asterisk recognizes certain characters to interpret special meaning when doing a pattern match. These characters include:

X - matches any digit from 0-9
Z - matches any digit from 1-9
N - matches any digit from 2-9
[1237-9] - matches any digit or letter in the brackets (ex. 1,2,3,7,8,9)
. - wildcard match which matches one or more characters

The following example shows how you would match a 4 digit extension number from 0000 through 9999. Make note of the _ underscore in front of the XXXX as this tells Asterisk that we are pattern matching and not to interpret the extension literally.

exten => _XXXX,1,NoOp()

Making use of the ${EXTEN} channel variable

The ${EXTEN} channel variable contains the extension number that was dialed. We use the ${EXTEN} variable along with pattern matching to help reduce the the number of lines we require to dial. This allows us to simply match a number using a call pattern, and then use the specific number dialed. The following examples shows a simple pattern match and use of ${EXTEN}.

```
   [globals]
   RECEPTIONIST=Zap/1
; Assign 'Zap/1' (FXS) to the global variable ${RECEPTIONIST}

   JOHN=SIP/1001
     ; Assign 'SIP/1001' to the global variable ${JOHN}

   MARY=SIP/1002
     ; Assign 'SIP/1002' to the global variable ${MARY}


   [incoming]
   exten => s,1,Answer()
   exten => s,2,Background(current-movies)
   exten => s,3,Hangup()
   exten => 1,1,Playback(movie1)
   exten => 1,2,Goto(incoming,s,1)
   exten => 2,1,Playback(movie2)
   exten => 2,2,Goto(incoming,s,1)
   exten => 0,1,Dial(${RECEPTIONIST})

   [internal]
   exten => _1XXX,1,Dial(SIP/${EXTEN})
; replace separate
Dial()
 statements with a single pattern matching statement
```

Note: Do not separate your pattern matches with hyphens (-) as this could possibly lead to errors.

The outgoing context contains a simple pattern matching example which would allow us to call a 4 digit number starting with **1** . We follow the same format for other extension lines by following the pattern matching line with a priority and application. Instead of specifically passing the number to the `Dial()` application we wish to call, we can make use of the ${EXTEN} channel variable. For example, if we were to dial 1001, then our pattern would match that number. The ${EXTEN} variable would then contain the number dialed, and be passed to the `Dial()` application. We have effectively replaced the two separate `Dial()` statements with a single line. All extension numbers in the range of 1000 -> 1999 would be matched by this line and we no longer need to add a separate line for each extension available to our PBX.

Now that we understand how pattern matching and the ${EXTEN} channel variable works, let's create an outgoing context to allow outbound dialing. Again, we'll assume that our outbound channel has been setup, and is called Zap/2.

```
[globals]
RECEPTIONIST=Zap/1
JOHN=SIP/1001
MARY=SIP/1002
LOCALTRUNK=Zap/2

[incoming]
exten => s,1,Answer()
exten => s,2,Background(current-movies)
exten => s,3,Hangup()
exten => 1,1,Playback(movie1)
exten => 1,2,Goto(incoming,s,1)
exten => 2,1,Playback(movie2)
exten => 2,2,Goto(incoming,s,1)
exten => 0,1,Dial(${RECEPTIONIST})

[internal]
exten => _1XXX,1,Dial(SIP/${EXTEN})
; replace separate
Dial()
statements with a single pattern matching statement
```

```
[outgoing]
ignorepat => 9
exten => _9NXXNXXXXXX,1,Dial(${LOCALTRUNK}/${EXTEN:1})
exten => _9NXXNXXXXXX,2,Playback(invalid)
exten => _9NXXNXXXXXX,3,Hangup
```

Note: The ignorepat statement allows us to keep playing dialtone to the user even after they have dialed 9 to access the outside line. Without it, there would be no dial tone after dialing 9.

We have now created a new context called "outgoing" which we can then use to give access to outbound phone lines.

The first line of our new context contains a new command called **ignorepat** which stands for ignore pattern. This will allow us to setup our outgoing context so that the user is required to dial **9** before dialing their outside phone number. Once the user presses **9** the dial tone will continue to be played to the caller. Without it, there would be no dial tone after dialing the number for accessing the outline line.

Our next line contains the `Dial()` application which will dial out through our ${LOCALTRUNK} interface to the number contained within the ${EXTEN} variable. The ${EXTEN} variable contains a :1 which is used to strip off the first digit that we dialed. This is because we do not want to pass the number 9 along with our phone number. For example, if we wanted to dial the phone number 123-555-1234, then we would dial 91235551234. Without the ${EXTEN:1}, then entire string would be passed to the `Dial()` application causing an error (or, the wrong number to be dialed). The ${EXTEN:1} causes only the 1235551234 part to be passed, removing the leading **9** .

If the `Dial()` application does not complete successfully, then the next priority will be executed. This will playback the `invalid` file to the user to let them know their call could not be completed. After the file is played, the channel is hung up.

You may have noticed that we have no way to test this context as nothing has access to it. To do this, we must make use of the **include** statement which will be explained in the next section.

Linking Contexts with Includes

Asterisk gives us the ability to use a context within another context. This is most often used to limit and grant access to different sections of the PBX. With the use of the **include** statement we can control who has access to toll services.

The include statement takes the form:

```
   include =>
context
```

When we include other contexts within our current context, we have to be aware of the order we are including them. Asterisk will first try and match the extension in the current context. If unsuccessful, it will then try the first included context, following down the list in order.

Our dialplan that we've been building now has an *outgoing* context, but as it sits we can't access it. In order to give our internal extensions the ability to access the outgoing context, we have to include it.

```
   [globals]
   RECEPTIONIST=Zap/1
; Assign 'Zap/1' (FXS) to the global variable ${RECEPTIONIST}

   JOHN=SIP/1001
     ; Assign 'SIP/1001' to the global variable ${JOHN}

   MARY=SIP/1002
   ; Assign 'SIP/1002' to the global variable ${MARY}

   LOCALTRUNK=Zap/2
 ; Assign 'Zap/2' (FXO) to the global variable ${LOCALTRUNK}


   [incoming]
   exten => s,1,Answer()
```

```
   exten => s,2,Background(current-movies)
   exten => s,3,Hangup()
   exten => 1,1,Playback(movie1)
   exten => 1,2,Goto(incoming,s,1)
   exten => 2,1,Playback(movie2)
   exten => 2,2,Goto(incoming,s,1)
   exten => 0,1,Dial(${RECEPTIONIST})

   [internal]
   exten => _1XXX,1,Dial(SIP/${EXTEN})
; replace separate
Dial()
 statements with a single pattern matching statement

   include => outgoing
; include the outgoing context


   [outgoing]
   ignorepat => 9
   exten => _9NXXNXXXXXX,1,Dial(${LOCALTRUNK}/${EXTEN:1})
   exten => _9NXXNXXXXXX,2,Playback(invalid)
   exten => _9NXXNXXXXXX,3,Hangup
```

In our above dialplan example we've added a single line to the internal context to include the outgoing context into it. This will give our internal users the ability to dial an outside line by dialing **9** plus the 10 digit phone number. For now we have limited our users simply to dialing local numbers but we could expand this example to allow long distance dialing as well by creating a new context for long distance dialing and including it in our internal context.

We can see that we have limited outgoing calls simply to our internal extensions; no one reaching the incoming context will have access to our outgoing context.

Some Other Special Extensions

's' - start
'i' - invalid
't' - timeout
'h' - hangup
'T' - Absolute Timeout

The *start* extension is for most calls that are initiated with no other known information.

Invalid is for when Asterisk has determined that the input from the call is not valid for the current context. You may wish to play a prompt explaining the extension was invalid, and then send the call back to the extension that contains the menu prompts.

Timeout is for when a user is presented with a menu and they do not respond. In the timeout extension you will want to decide if you wish to repeat your menu, or just send the call to a hangup so as to free up the line.

Hangup is where calls will go to when hangup is detected, or where you can send calls that you want to hangup on.

> **Warning**
>
> There are currently some problems to be aware of when using the 'h' extension. Specifically, the variables about the call are lost as the information is destroyed with the channel.

Absolute Timeout is used when a call is being terminated for exceeding an Absolute Timeout variable set. Be aware of the case difference from the normal timeout. This can be used to warn a user that they exceeded some allowable limit. Or it could be used to request someone to try calling back later if they waited in a queue too long. Essentially it should notify the caller that they are being disconnected so as not to leave them with the impression they had been cut off unintentially.

Creating Prompts

While Asterisk comes with many prompts which we can use to create our dialplans, it is impossible to include all possible prompts and menus. Asterisk includes a very handy application which we can use to record our own prompts. Obviously enough, the application is called `Record()`.

Use of the `Record()` Application

The `Record()` application can be used to record the audio traffic on a channel. We can make use of this application to record custom prompts for use within our dialplan.

Lets create a new context which will let us change the prompts for the movie theatre's in our example dialplan. By pressing ***1** or ***2** , we can re-record the prompts for theatres one and two respectively.

Example 5-1. Custom Prompt Creation Context

```
[prompts]
exten => *1,1,Answer()
exten => *1,2,Record(movie1:gsm)
exten => *1,3,Playback(movie1)
exten => *1,4,Hangup()

exten => *2,1,Answer()
exten => *2,2,Record(movie2:gsm)
exten => *2,3,Playback(movie2)
exten => *2,4,Hangup()
```

We can give access to this context by including it in our **internal** context. Our **internal** context will look like the following.

```
[internal]
exten => _1XXX,1,Dial(SIP/${EXTEN})
; replace separate
Dial()
statements with a single pattern matching statement

include => outgoing
; allow internal extens to make outgoing calls

include => prompts
; allow internal extens to record prompts
```

By including the **prompts** context within our **internal** context, we have given access to the ***1** and ***2** extensions to allow us to change our `movie1` and `movie2` prompts.

Use of the `Authenticate()` Application

As we showed in the previous section, we can record our own custom prompts with Asterisk. But perhaps we would like to add a layer of security and require someone to be authenticated before they can change the prompts. A caller can be authenticated before they have access to change anything with the use of the `Authenticate()` application. The following examples will show you how we can make use of this in combination with our `Record()` application.

We are going to edit our previous example slightly to allow a bit more flexibility to our dialplan. We'll create a special features context which we will access by pressing `*` . Asterisk will then ask us to authenticate before allowing us access to the special features menu. Once authenticated, then we can change the movie theatre prompts by pressing the corresponding theatre number.

Example 5-2. Using the `Authenticate()` Application

```
[special]
exten => *,1,Answer()
exten => *,2,Authenticate(1234)
exten => *,3,Goto(prompts,s,1)

[prompts]
exten => s,1,Answer()
exten => s,2,Background(
```

Conclusion

By this point it must be obvious that we have simply scratched the surface of dialplan creation within Asterisk. However, you now have enough knowledge to go out and learn the many applications that Asterisk has built into it to expand your dialplan. You should now understand many of the fundamental applications and be well on your way to creating powerful dialplans. Spend more time playing with the applications built into Asterisk and expanding upon the dialplan we've created in this chapter by utilizing the basic skills you have aquired from this book. Asterisk is a wonderful and ever evolving application. We hope you have learned much and continue developing with the Asterisk platform.

Colophon

This document was written as an introduction to the Asterisk PBX system. Topics include Installation, OS Preparation, Channel Setup, Compilation and Introduction to Dialplans.