O'REILLY®

Fifth Edition
Covers Asterisk 16

Asterisk
The Definitive Guide

Open Source Telephony for the Enterprise

Jim Van Meggelen,
Russell Bryant
& Leif Madsen

O'REILLY®

Asterisk: The Definitive Guide

Design a complete Voice over IP (VoIP) or traditional PBX system with Asterisk, even if you have only basic telecommunications knowledge. This bestselling guide makes it easy with a detailed roadmap that shows you how to install and configure this open source software, whether you're upgrading your existing phone system or starting from scratch.

Ideal for Linux administrators, developers, and power users, this updated fifth edition shows you how to set up VoIP-based private telephone switching systems within the enterprise. You'll get up to speed on the features in Asterisk 16, the latest long-term support release from Digium. This book also includes new chapters on WebRTC and the Asterisk Real-time Interface (ARI).

- **Discover how WebRTC provides a new direction for Asterisk**
- **Gain the knowledge to build a simple but complete phone system**
- **Build an interactive dialplan, using best practices for Asterisk's advanced features**
- **Learn how ARI has emerged as the API of choice for interfacing web development languages with Asterisk**

From the Foreword

"This book will help you see the most modern face of Asterisk and how to better utilize it in your telecommunications infrastructure."

—**Matt Fredrickson**
Director of Asterisk Engineering, Sangoma/Digium

Jim Van Meggelen has nearly 30 years of enterprise telecom experience. He is a founding partner of ClearlyCore Inc.

Russell Bryant, a Distinguished Engineer at Red Hat, works on cloud infrastructure projects. Before Red Hat, he worked on the Asterisk project for seven years.

Leif Madsen is the Cloud Service Assurance Architect within the CloudOps team at Red Hat.

NETWORKING / SYS ADMIN

US $64.99 CAN $85.99

ISBN: 978-1-492-03160-4

Twitter: @oreillymedia
facebook.com/oreilly

FIFTH EDITION

Asterisk: The Definitive Guide

Jim Van Meggelen, Russell Bryant, and Leif Madsen

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Asterisk: The Definitive Guide
by Jim Van Meggelen, Russell Bryant, and Leif Madsen

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Rachel Roumeliotis
Development Editor: Jeff Bleiel
Production Editor: Kristen Brown
Copyeditor: Dwight Ramsey
Proofreader: Rachel Monaghan

Indexer: Judith McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

July 2019: Fifth Edition

Revision History for the Fifth Edition
2019-06-21: First Release

See *https://www.oreilly.com/catalog/errata.csp?isbn=0636920140610* for release details.

978-1-492-03160-4

[LSI]

Table of Contents

Foreword

When contemplating the foreword for every edition of this book, we always had more people we'd like contributions from than pages we could spare. In this fifth edition, we've again asked a select group of people from the Asterisk community to write a few words about Asterisk from their perspective.

Joshua Colp (Senior Software Developer, Sangoma/Digium)

Over 15 years ago, I downloaded Asterisk onto my laptop and placed my first VoIP call using IAX2 to the Digium PBX. I held my breath in anticipation, waiting to hear a voice, until finally the sound of Allison came out of my laptop. At that point I knew there was something special to Asterisk. It lit this spark of interest and imagination in me: my laptop had actually placed a call! The realization that with only a little effort I could take calls and do with them what I wanted was addictive and exciting—a sentiment shared by many to this day.

Asterisk today is vastly different from how it was during that time. In the past, it was primarily focused on being a PBX. It had all of the features and continued to gain new ones to propel it further into that area. Over time, however, the project has evolved to one where Asterisk is a toolkit that can be used alone or in combination with other projects to build things. It's there to spark the question of "Can I do this?" in your mind and allow you to see it through.

This simple question is what drives many of the decisions made about Asterisk and its direction. "Is this right for the users?", "Is this what people truly need?," "Does this break things?," and "Can they build what they want with this?" Together, these questions help ensure that people can realize their ideas. This is what excites me about Asterisk today—seeing people use the tools to create something new without hindrance.

I think that, going forward, this will continue for Asterisk. It will continue to add new tools and functionality to provide greater flexibility and options for those building things, while respecting its legacy and how users already use it. It will continue to be part of bigger and better solutions, some of which may not even come to mind now. We've only taken a few steps forward and have many to go.

I challenge new and old users of Asterisk alike to revisit what Asterisk can do, to learn new features that have been added, and to build something new and exciting out of your normal skill set. If you hit a roadblock where Asterisk can't do what you need, then participate in the project and contribute. Help others who may be trying to do the same thing. Become not just someone who uses Asterisk, but also someone who helps others realize their dream.

Dan Jenkins (Founder, Nimble Ape Ltd)

Asterisk was my first foray into the open source telephony world, and as a web developer, I found it very different from what I was used to, coming from the web industry. The Asterisk project has moved on since then, and now the project incorporates many APIs and technologies that a typical web developer has come to expect. The inclusion of WebRTC and Asterisk's REST interface is vital for integration from developers used to building for the web platform. Asterisk is what I eventually ended building a business around—it is truly a remarkable piece of software and has a brilliant community of people who both use and improve it. It's been my pleasure being a part of this community and proofreading this book for the future community.

Joyce Wilmot (Senior Web Developer)

I was introduced to Asterisk in 2012 when I was working for Voicenation, a company that provides live answering service 24/7/365 for thousands of customers. At the time, the call center was quickly outgrowing the third-party software they were using. Unable to find a flexible and cost-effective solution for their quickly expanding call center, Voicenation decided that they needed to create their own call center software. I was tasked with creating this software, which started my journey with Asterisk. What started as a monumental task (since I did not have prior IP telephony experience) quickly became a fascination with Asterisk as I discovered how it simplified our setup without sacrificing power and flexibility.

Fast-forward nine years and tens of millions of calls later, and Asterisk still faithfully and reliably runs our call center. This was my first exposure to open source software. Asterisk is obviously an open source success story that illustrates how open source software fuels entrepreneurism—and how entrepreneurism, in return, fuels development and enhancement of the open source software. I'm thrilled to be part of that

cycle, and look forward to being part of the community as Asterisk continually evolves to keep up with the ever-changing world of telecommunications.

Matt Florell (Founder, VICIdial)

My first exposure to open source telephony back in 2001 was actually not with Asterisk. It was with a different software package, one that took me a couple of months to get working, using a simple IVR to log contact requests for my employer at the time. It was not an easy system to work with or modify, so I didn't do much else with it beyond that first IVR project. Two years later, I had a request from a client to build a much more complex telephony system, one that would require user interactions through a computer. I knew the platform I had been using wasn't going to work for a project like this, so I looked around at both commercial and open source options. That's when I learned about Asterisk, which looked like it could be an ideal platform for this project. I bought a T1 card with which to do some testing, and within two hours of its arrival, I had configured it and I was able to replicate the old project that had taken me two months to build. After that, I was hooked. The VICIdial Open-Source Contact Center project grew out of that project; to date, over 100,000 Asterisk systems have been installed as a part of VICIdial clusters, and those are just the ones we know about.

Asterisk was very different from the mostly web-based open source packages that I had worked with in the past, and it had quite a few quirks and bugs in the earlier days that you had to work around (sometimes in pretty creative ways). But our more recent experiences with the Asterisk 13 branch have shown significant improvements in both capacity and stability, compared to earlier branches. There have also been many new features added that have allowed us to add new functionality to our VICIdial package. Two of those are the ability to pause call recordings and the addition of several layers of new SIP carrier logging.

Back in 2003 when I started using Asterisk, there were no real "releases." You had to find a stable build from one of the recent CVS revisions and test it out. As time went on, the development and maintenance of the different branches became much more stable, and the use of Asterisk in production systems all over the world skyrocketed. Today, Asterisk is the telephony core of thousands of different service offerings, with billions of phone calls a day being placed through them. It is being installed on wide varieties of hardware, from tiny embedded systems to server farms with hundreds of high-powered machines. There are now millions of people who use Asterisk every day who have no idea that they are interacting with a piece of open source software.

Among our client base alone, we have several Fortune 500 companies, as well as school districts, social clubs, political organizations, municipal emergency services organizations, and of course, thousands of different types of commercial operations. While the low acquisition cost is a common reason given for going with an Asterisk-

based solution, we often hear that the fact that it is open source is a big plus, as well as there being no possibility of vendor lock-in. One of our larger clients even cited their use of open source telephony software as a "distinct strategic advantage" over their competitors because of the flexibility of the systems and their ability to self-manage them without having to rely on outside vendors. From what I've seen so far, the future of Asterisk is an ever-growing installed base and continued enhancements. I look forward to working with it for another 16 years, at least.

Matt Fredrickson (Director of Asterisk Engineering, Sangoma/Digium)

I have had the opportunity to work with Asterisk for the last 18 years, and have seen it grow from a small project with one or two people into something that has a life of its own with hundreds of contributors. It's amazing to see the number of different places it has disrupted traditional telecom—at home, in the office, and in the enterprise. As traditional communication patterns shift, the Asterisk project continues to be in the place where it does best—bridging old forms of communication with the new, and pushing the boundaries of what can be done with the new. This book will help you see the most modern face of Asterisk and how to better utilize it in your telecommunications infrastructure. Huge thanks goes to Jim Van Meggelen for all the hard work on putting this most current edition together.

Preface

This is a book for anyone who uses Asterisk.

Asterisk is an open source, converged telephony platform, which is designed primarily to run on Linux. Asterisk combines more than 100 years of telephony knowledge into a robust suite of tightly integrated telecommunications applications. The power of Asterisk lies in its customizable nature, complemented by unmatched standards compliance. No other private branch exchange (PBX) can be deployed in so many creative ways.

Applications such as voicemail, hosted conferencing, call queuing and agents, music on hold, and call parking are all standard features built right into the software. Moreover, Asterisk can integrate with other business technologies in ways that closed, proprietary PBXs can scarcely dream of.

Asterisk can appear quite daunting and complex to a new user, which is why documentation is so important to its growth. Documentation lowers the barrier to entry and helps people contemplate the possibilities.

Produced with the generous support of O'Reilly Media, *Asterisk: The Definitive Guide* is the fifth edition of what was formerly called *Asterisk: The Future of Telephony*.

This book was written for, and by, members of the Asterisk community.

Audience

This book is intended to be gentle toward those new to Asterisk, but we assume that you're familiar with basic Linux administration, networking, and other IT disciplines. If not, we encourage you to explore the vast and wonderful library of books that O'Reilly publishes on these subjects. We also assume you're fairly new to telecommunications (both traditional switched telephony and the new world of Voice over IP).

However, this book will also be useful for the more experienced Asterisk administrator. We ourselves use the book as a reference for features that we haven't used for a while.

Software

This book is focused on documenting Asterisk version 16; however, many of the conventions and much of the information in this book is version-agnostic. Linux is the operating system we have run and tested Asterisk on, and we have documented installation instructions for CentOS (Red Hat Enterprise Linux, or RHEL).

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, file extensions, pathnames, directories, and package names, as well as Unix utilities, commands, modules, parameters, and arguments.

`Constant width`
: Used to display code samples, file contents, command-line interactions, database commands, library names, and options.

`Constant width bold`
: Indicates commands or other text that should be typed literally by the user. Also used for emphasis in code.

`Constant width italic`
: Shows text that should be replaced with user-supplied values.

`[ Keywords and other stuff ]`
: Indicates optional keywords and arguments.

`{ choice-1 | choice-2 }`
: Signifies either *`choice-1`* or *`choice-2`*.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

O'Reilly Online Learning

For almost 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/asterisk_tdg_5E*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments from Jim Van Meggelen

To David Duffett, thanks for the chapter on internationalization, which properly looks at this technology from a more global perspective.

Thanks to Leif Madsen, Jared Smith, and Russell Bryant, for your contributions to the previous editions of this book. It was fun flying solo, but I can't deny I missed you guys!

Specific thanks to Matt Fredrickson and Matt Jordan of Digium, who generously shared their time and knowledge with me, and without whom I would have been lost. Thanks guys!

Thanks to my editor, Jeff Bleiel, for keeping me on track and helping me make important decisions about the content and pacing of the book.

Also thanks to the rest of the unsung heroes in O'Reilly's production department. These are the folks that take a book and make it an *O'Reilly book*.

Thanks especially to Joyce Wilmot and Dan Jenkins, my technical review team, for taking the time to work through the book and provide essential feedback.

Thomas Cameron of RedHat generously shared his knowledge of Selinux with me, and helped to demystify a Linux component that is too often left disabled.

Everyone in the Asterisk community also needs to thank the late Jim Dixon for creating the first open source telephony hardware interfaces, starting the revolution, and giving his creations to the community at large.

Finally, and most importantly, thanks go to Mark Spencer, the original author of Asterisk and founder of Digium, for Asterisk, for Pidgin, and for contributing his creations to the open source community. Asterisk is your legacy!

CHAPTER 1
A Telephony Revolution

We are what they grow beyond. That is the true burden of all masters.
—Jedi Master Yoda

When we first set out in 2004 to write a book about Asterisk (15 years ago as of this edition!), we confidently predicted that Asterisk would fundamentally change the telecommunications industry. Today, the revolution we predicted is a part of history. Asterisk has been the most successful private branch exchange (PBX) in the world for several years now, and is an accepted technology within the telecommunications industry.

The revolution—as necessary as it was to the telecommunications industry of that time—has tailed off significantly simply because the methods by which people like to communicate have changed. Whereas 25 years ago phone calls were the preferred way to converse across distances, the current trend is to send messages or conduct video-chat conference calls. The phone call is seen as a bit of a dead thing, especially by up-and-coming generations. We're not quite ready for a funeral just yet.

Asterisk remains a powerful technology, and we believe it is still one of the best hopes for any sort of sensible integration between telecom and all the other technologies businesses might want to interconnect with. It will need to find its place within a communications ecosystem that no longer places telephone calls in a place of importance. Our expectation is that WebRTC, which promises to commoditize web-based communications,[1] will emerge as a replacement for all the copycat, closed, and proprietary "collaboration" products currently flooding (and confusing) the market. Asterisk can play a role in this new future, and the Asterisk community has willingly and enthusiastically taken on this new concept. So, maybe you're being told that voice

1 And more, perhaps, given that WebRTC is revolutionizing native apps too!

is dead, but anyone who's paid attention to any science fiction of any kind knows that being able to talk to each other across long distances is not going to be the sole domain of those who type on keyboards. Humans like to talk, and we'll continue to find ways to do so.

There also exists, it must be noted, a massive generation of people whose memories predate the internet, and for these folks the telephone is still a very useful technology. If one wishes to do business with them, one had better do a good job of handling telephone calls. These folks are retiring from the workforce, but their wallets still carry a lot of clout. Perhaps the PBX is a dying thing, but its tail is very long.

In this book, we're going to explore the nuts and bolts of Asterisk. It is a flexible, open, standards-compliant toolkit, which we believe is still very relevant to businesses today, and will remain useful for many years to come. The power of Asterisk lies in its flexibility. It has proven to be very useful at tying various types of communications technologies together, and if it is to have any sort of future, it will need to continue to do so. Newer technologies such as WebRTC offer all sorts of possibilities for the future of communication, and the Asterisk community is very focused on this paradigm shift.

The remarkable flexibility of Asterisk comes with a price: it is not a simple system to learn or configure. This is not because it's illogical, confusing, or cryptic; on the contrary, it is very sensible and practical. People's eyes light up when they first see an Asterisk dialplan and begin to contemplate the possibilities. But when there are literally thousands of ways to achieve a result, the process naturally requires extra effort. Perhaps it can be compared to building a house: the components are relatively easy to understand, but a person contemplating such a task must either a) enlist competent help or b) develop the required skills through instruction, practice, and a good book on the subject.

Asterisk and VoIP: Bridging the Gap Between Traditional and Network Telephony

It sometimes seems that we've forgotten that the purpose of the telephone is to allow people to communicate. It is a simple goal, really, and it should be possible for us to make it happen in far more flexible and creative ways than are currently available to us. New technologies always seek to dominate the market with a proprietary offering. Few succeed. Communications technologies need to interoperate, and technologies such as Asterisk lower the barriers to entry for those wishing to innovate.

It is for this reason—communication—that we believe a future still exists for open source telephony projects such as Asterisk. Yes, people might not want to make "phone calls" anymore, but we believe there will still be value found in conversations.

The technologies that can facilitate those conversations may evolve in seemingly radical ways, yet the underlying desire to communicate remains the same.

Asterisk is plugged into the future, and it has a long track record of successfully integrating communications technologies.

The Zapata Telephony Project

When the Asterisk project was started (in 1999), there were other open source telephony projects in existence. However, Asterisk, in combination with the Zapata Telephony Project, was able to provide public switched telephone network (PSTN) interfaces, which represented an important milestone in transitioning the software from something purely network-based to something more practical in the world of telecom at that time, which was PSTN-centric.

The Zapata Telephony Project was conceived of by Jim Dixon, a telecommunications consulting engineer who was inspired by the incredible advances in CPU speeds that the computer industry has now come to take for granted. Dixon's belief was that far more economical telephony systems could be created if a card existed that had nothing more on it than the basic electronic components required to interface with a telephone circuit. Rather than having expensive components on the card, digital signal processing (DSP)[2] would be handled in the CPU by software. While this would impose a tremendous load on the CPU, Dixon was certain that the low cost of CPUs relative to their performance made them far more attractive than expensive DSPs, and, more importantly, that this price/performance ratio would continue to improve as CPUs continued to increase in power.

Like so many visionaries, Dixon believed that many others would see this opportunity, and that he merely had to wait for someone else to create what to him was an obvious improvement. After a few years, he noticed that not only had no one created these cards, but also it seemed unlikely that anyone was ever going to. At that point it was clear that if he wanted a revolution, he was going to have to start it himself. And so the Zapata Telephony Project was born:

> Since this concept was so revolutionary, and was certain to make a lot of waves in the industry, I decided on the Mexican revolutionary motif, and named the technology and organization after the famous Mexican revolutionary Emiliano Zapata. I decided to call the card the "tormenta" which, in Spanish, means "storm," but contextually is usually used to imply a big storm, like a hurricane or such.

2 The term DSP also means digital signal processor, which is a device (usually a chip) that is capable of interpreting and modifying signals of various sorts. In a voice network, DSPs are primarily responsible for encoding, decoding, and transcoding audio information. This can require a lot of computational effort.

Perhaps we should be calling ourselves Asteristas. Regardless, we owe Jim Dixon a debt of thanks, partly for thinking this up and partly for seeing it through, but mostly for giving the results of his efforts to the open source community. As a result of Jim's contribution, Asterisk's PSTN engine came to be. And thanks to this marrying of VoIP and PSTN, the open source telecom revolution was born!

Over the years, the Zapata Telephony interface in Asterisk has been modified and improved. The Digium Asterisk Hardware Device Interface (DAHDI) telephony interface in use today is the offspring of Jim Dixon's contribution.

Massive Change Requires Flexible Technology

Every PBX in existence suffers from shortcomings. No matter how fully featured it is, something will always be left out, because even the most feature-rich PBX will always fail to anticipate the creativity of the customer. A small group of users will desire an odd little feature that the design team either did not think of or could not justify the cost of building, and, since the system is closed, the users will not be able to build it themselves.

If the internet had been thusly hampered by regulation and commercial interests, it is doubtful that it would have gained the wide acceptance it currently enjoys. The openness of the internet meant that anyone could afford to get involved. So, everyone did. The tens of thousands of minds that collaborated on the creation of the internet delivered something that no corporation alone ever could have.[3]

As with many other open source projects (such as Linux and so much of the critical software running the internet), the development of Asterisk was fueled by the dreams of folks who knew that there had to be something more than what traditional industries were producing. These people knew that if one could take the best parts of various PBXs and separate them into interconnecting components—akin to a boxful of LEGO bricks—one could begin to conceive of things that would not survive a traditional corporate risk-analysis process.

Asterisk itself has become the basis of many massively productized creations. And yet, under the hood, the soul of that open source project still remains.

Asterisk: The Hacker's PBX

Asterisk is the ultimate hacker's PBX. The term *hacker* has, of course, been twisted by the mass media into meaning "malicious cracker" to the uneducated. This is unfortu-

3 We realize that the technology of the internet formed out of government and academic institutions, but what we're talking about here is not the technology of the internet so much as the cultural phenomenon of it, which exploded in the early '90s.

nate, because the term actually existed long before the media corrupted its meaning. Hackers built the networking engine that is the internet. Hackers built the Apple Macintosh and the Unix operating system. Hackers are also building your next telecom system. Yes, some of these folks are malicious, but the minds that steer the development of Asterisk are well aware of this, and you'll find that Asterisk allows you to build a system that's far more capable of rapidly responding to security threats. Open source software doesn't hide its faults behind corporate spin departments. The dirt gets dragged out into the open where it can be dealt with. Rather than being constricted by the dubious and often poor security of closed systems, the Asterisk community quickly responds to changing trends in security, and you'll be able to fine-tune your telephone system in response to both corporate policy and industry best practices.

Like other open source systems, Asterisk will be able to evolve into a far more secure platform than any proprietary system, not in spite of its hacker roots, but rather because of them.

Asterisk: The Professional's PBX

Asterisk is an enabling technology, and as with Linux, it will become increasingly rare to find an enterprise that is not running some version of Asterisk, in some capacity, somewhere in the network, solving a problem as only Asterisk can. You're already using Asterisk, even if you don't know it.

The Asterisk Community

There's no sense beating around the bush: the Asterisk community is a shadow of its former self. A dozen years ago, Asterisk was just about the coolest thing in open source. Today, most enthusiasts have moved on. What remains, however, is an experienced and battle-tested community of professionals, who have been there and done that.

Do not expect a team of people willing to work for free on your projects. The price of entry to this community is a personal commitment to skills development. If you bring a sense of entitlement to this community, you will not enjoy the responses. If, however, you bring curiosity and enthusiasm and a willingness to dive in, get your hands dirty, and do the work, you will find a community more than willing to share their hard-won knowledge with you.

The following are some of the places the Asterisk community hangs out.

Asterisk's Discourse-Based Community Site

Asterisk moved its official forums to *https://community.asterisk.org/* in 2015. This appears to be the most active community right now, and the signal-to-noise ratio is

excellent. The Digium folks do a good job of moderating this, and several of their senior and experienced people are actively involved.

Bear in mind that this is not like paid support. You are expected to do the work yourself, but you can expect to get some good quality advice here, which can help to steer you in the right direction.

The Asterisk Mailing Lists

The activity on these lists has been reduced to a trickle (down from hundreds of messages per day to maybe a dozen threads per month). They are probably most useful as an historical archive, but may be worth searching through when you're working on an intractable problem. Of the mailing lists you will find at lists.digium.com, these two are likely to be the most useful:

Asterisk-Users
: This list is a shadow of its former self. Whereas it used to generate several hundred messages per day, most of this traffic has moved to Digium's Asterisk Community site (above).

Asterisk-Dev
: The Asterisk developers hang out here. The purpose and focus of this list is the discussion of developing the software that is Asterisk, and participants vigorously defend that purpose. Expect a lot of heat if you post anything to this list not specifically relating to the programming or development of the Asterisk code base. General coding questions (such as queries on interfacing with AGI or AMI) should be directed to the *Asterisk-Users* list.

The Asterisk-Dev list is not second-level support! If you scroll through the mailing list archives, you'll see this is a strict rule. The Asterisk-Dev mailing list is about discussion of core Asterisk development, and questions about interfacing your external programs via AGI or AMI should be posted on the *Asterisk-Users* list.

Asterisk Wiki Sites

This isn't really a community hangout, but it deserves a mention. Digium maintains a wiki for Asterisk at *wiki.asterisk.org*. This site is constantly kept up to date by the Digium team, and automated scripts export the XML-based documentation from the Asterisk source into the wiki itself, all of which helps to ensure that the data you're reading is an accurate representation of the world.

An older wiki exists at *www.voip-info.org*, which is these days somewhat of an historical curiosity, and a source of much enlightenment and confusion. While there is a

massive amount of information contained here, much of it is out of date. We include reference to it here simply because you're likely to land on it one day and think you've hit the mother lode, but what you've actually found is more akin to a museum of historical oddities: fascinating, but not necessarily relevant.

The IRC Channels

The Asterisk community maintains Internet Relay Chat (IRC) channels on *irc.freenode.net*. The two most active channels are *#asterisk* and *#asterisk-dev*.[4] To cut down on spambot intrusions, both of these channels require registration to join. To register, run `/msg nickserv help` when you connect to the service via your favorite IRC client.

Conclusion

So where to begin? Well, when it comes to Asterisk, there is far more to talk about than we can fit into one book. This book can only lay down the basics, but from this foundation you will be able to come to an understanding of the concept of Asterisk—and from that, who knows what you will build?

4 The *#asterisk-dev* channel is for the discussion of changes to the underlying code base of Asterisk and is also not second-tier support. Discussions related to programming external applications that interface with Asterisk via AGI or AMI are meant to be in *#asterisk*.

CHAPTER 2
Asterisk Architecture

First things first, but not necessarily in that order.
—Doctor Who

Asterisk is very different from other, more traditional, PBXs in that the dialplan in Asterisk treats all incoming channels in essentially the same manner, rather than separating them into stations, trunks, peripheral modules, and so forth.

In a traditional PBX, there is a logical difference between stations (telephone sets) and trunks (resources that connect to the outside world). This limitation makes creative routing in traditional PBXs very difficult or impossible.

Asterisk, on the other hand, does not have an internal concept of trunks or stations. In Asterisk, everything that comes into or goes out of the system passes through a channel of some sort. There are many different kinds of channels; however, the Asterisk dialplan handles all channels in a similar manner, which means that, for example, an internal user can exist on the end of an external trunk (e.g., a cell phone) and be treated by the dialplan in exactly the same manner as that user would be if they were on an internal extension. Unless you have worked with a traditional PBX,[1] it may not be immediately obvious how powerful and liberating this is. Figure 2-1 illustrates the differences between the two architectures.

1 A good indicator that you've worked with traditional PBXs is the presence of a large callus on your forehead, obtained from smashing your head against a brick wall too many times to count.

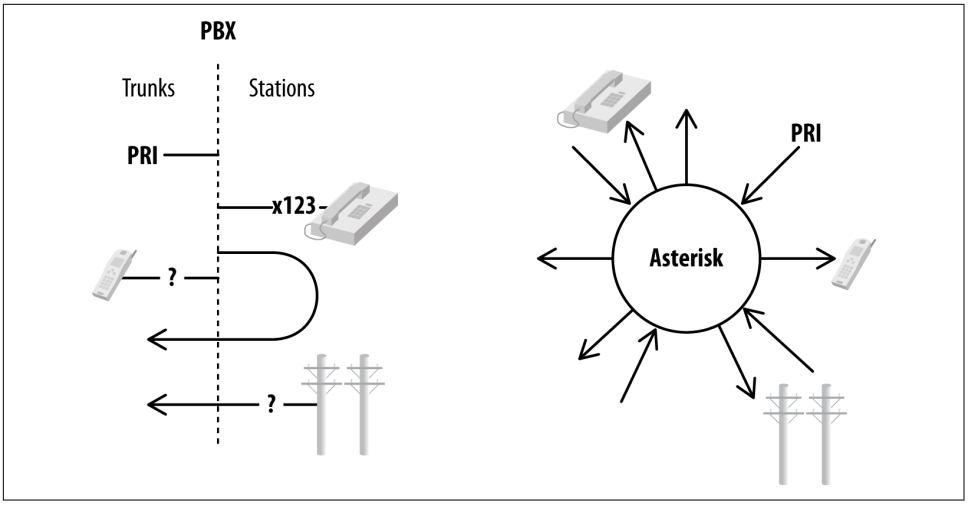

Figure 2-1. Asterisk versus PBX architecture

Modules

Asterisk is built on *modules*. A module is a loadable component that provides a specific functionality, such as a channel driver (for example, *chan_pjsip.so*), or a resource that allows connection to an external technology (such as *func_odbc.so*). Asterisk modules are loaded based on the parameters defined in the */etc/asterisk/modules.conf* file. We will discuss the use of many modules in this book, but at this point we just want to introduce the concept of modules, and give you a feel for the types of modules that are available.

It is actually possible to start Asterisk without any modules at all, although in this state it will not be capable of doing anything. It is useful to understand the modular nature of Asterisk in order to appreciate the architecture.

You can start Asterisk with no modules loaded by default and load each desired module manually from the console, but this is not something that you'd want to put into production; it would only be useful if you were performance-tuning a system where you wanted to eliminate everything not required by your specific application of Asterisk.

The types of modules in Asterisk include the following:

- Applications—The workhorses of the dialplan, such as `Dial()`, `Voicemail()`, `Playback()`, `Queue()`, and so forth
- Bridging modules—Mechanisms that connect channels (calls) to each other

- Call detail recording (CDR) modules
- Channel event logging (CEL) modules
- Channel drivers—Various connections into and out of the system; SIP (Session Initiation Protocol)messaging uses the PJSIP channel drivers
- Codec translators—Convert various codecs such as G729, G711, G722, Speex, and so forth
- Format interpreters—As above, but relating to files stored in the filesystem
- Dialplan functions—Enhance the capabilities of the dialplan
- PBX modules
- Resource modules
- Add-on modules
- Test modules

In the following sections we have curated a list of modules we feel are important enough to be discussed in this book. You'll find many other modules in the Asterisk download, but many older modules are either deprecated or have little or no support, and are therefore not recommended for production unless you have access to developers who can maintain them for you.

There is an official list of support status types included within `menuselect`.[2]

Applications

Dialplan applications are used in *extensions.conf* to define the various actions that can be applied to a call. The `Dial()` application, for example, is responsible for making outgoing connections to external resources and is arguably the most important dialplan application. The available applications are listed in Table 2-1.

Table 2-1. Popular dialplan applications

Name	Purpose
`app_authenticate`	Compares dual-tone multifrequency (DTMF) input against a provided string (password)
`app_cdr`	Writes ad hoc record to CDR
`app_chanspy`	Allows a channel to listen to audio on another channel
`app_confbridge`	Provides conferencing
`app_dial`	Used to connect channels together (i.e., make phone calls)
`app_directed_pickup`	Answers a call that's ringing at another extension

2 This is a command that is available as part of the installation process. We will discuss the use of menuselect in the installation chapter.

Name	Purpose
app_directory	Presents the list of names from *voicemail.conf*
app_dumpchan	Dumps channel variables to Asterisk command-line interface (CLI)
app_echo	Echos received audio back to source channel (can be helpful in demonstrating latency)
app_exec	Contains Exec(), TryExec(), and ExecIf(): executes a dialplan application conditionally
app_mixmonitor	Records both sides of a call (transmit and receive) and mixes them together into a single file
app_originate	Allows dialplan logic to originate a call (as opposed to a call coming in on a channel)
app_page	Creates multiple audio connections to specified devices for public address (paging)
app_parkandannounce	Enables automated announcing of parked calls
app_playback	Plays a file to the channel (does not accept input)
app_playtones	Plays pairs of tones of specified frequencies (DTMF mostly)
app_queue	Provides Automatic Call Distribution (ACD)
app_read	Requests input of digits from callers and assigns input to a variable
app_readexten	Requests input of digits from callers and passes call to a designated extension and context
app_record	Records received audio to a file
app_senddtmf	Transmits DTMF to calling party
app_stack	Provides GoSub(), GoSubIf(), Return(), StackPop(), LOCAL(), and LOCAL_PEEK()
app_stasis	Passes call control to an ARI application—many Asterisk developers use this one application, and from there handle all the rest of their development outside of the Asterisk dialplan
app_system	Executes commands in a Linux shell
app_transfer	Performs a transfer on the current channel
app_voicemail	Provides voicemail
app_while	Includes While(), EndWhile(), ExitWhile(), and ContinueWhile(); provides while loop functionality in the dialplan

Bridging Modules

Bridging modules perform the actual bridging of channels. These modules, listed in Table 2-2, are currently only used for (and are essential to) app_confbridge.

Table 2-2. Bridging modules

Name	Purpose
bridge_builtin_features	Performs bridging when utilizing built-in user features (such as those found in *features.conf*).
bridge_multiplexed	Performs complex multiplexing, as would be required in a large conference room (multiple participants). Currently only used by app_confbridge.
bridge_simple	Performs simple channel-to-channel bridging.
bridge_softmix	Performs simple multiplexing, as would be required in a large conference room (multiple participants). Currently only used by app_confbridge.

Call Detail Recording Modules

The CDR modules, listed in Table 2-3, are designed to facilitate as many methods of storing call detail records as possible. You can store CDRs to a file (the default), a database, Remote Authentication Dial In User Service (RADIUS), or *syslog*.

Call detail records are not intended to be used in complex billing applications. If you require more control over billing and call reporting, you will want to look at channel event logging, discussed next. The advantage of CDR is that it just works.

Table 2-3. Common call detail recording modules

Name	Purpose
`cdr_adaptive_odbc`	Allows writing of CDRs through ODBC framework with ability to add custom fields
`cdr_csv`	Writes CDRs to disk as a comma-separated values (CSV) file
`cdr_custom`	Writes CDRs to a CSV file, but allows addition of custom fields
`cdr_odbc`	Writes CDRs through ODBC framework
`cdr_syslog`	Writes CDRs to *syslog*

Channel Event Logging Modules

Channel event logging (CEL) provides much more powerful control over reporting of call activity. By the same token, it requires more careful planning of your dialplan, and by no means will it work automatically. Asterisk's CEL modules are listed in Table 2-4.

Table 2-4. Channel event logging modules

Name	Purpose
`cel_custom`	CEL to disk/file
`cel_manager`	CEL to AMI
`cel_odbc`	CEL to ODBC

Channel Drivers

Without channel drivers, Asterisk would have no way to make or receive calls. Each channel driver is specific to the protocol or channel type it supports (SIP, ISDN, etc.). The channel module acts as a gateway to the Asterisk core. Some of Asterisk's more popular channel drivers are listed in Table 2-5.

Table 2-5. Popular channel drivers

Name	Purpose
`chan_bridge`	Used internally by the `ConfBridge()` application; should not be used directly
`chan_dahdi`	Provides connection to PSTN cards that use DAHDI channel drivers
`chan_local`	Provides a mechanism to treat a portion of the dialplan as a channel
`chan_motif`	Implements the Jingle protocol, including the ability to connect to Google Talk and Google Voice; introduced in Asterisk 11
`chan_multicast_rtp`	Provides connection to multicast Realtime Transport Protocol (RTP) streams
`chan_pjsip`	Session Initiation Protocol (SIP) channel driver

Codec Translators

The codec[3] translators (often called *transcoders*) allow Asterisk to convert audio stream formats between calls. So if a call comes in on a PRI circuit (using G.711) and needs to be passed out a compressed SIP channel (e.g., using G.729, one of many codecs that SIP can handle), the relevant codec translator would perform the conversion.

> Codecs are complex algorithms that handle conversion of analog information (sound, in this case, but could be video as well) into a digital format. Many codecs provide compression and error correction as well, but this is not a requirement.

If a codec (such as G.729) uses a complex encoding algorithm, heavy use of transcoding can place a massive burden on the CPU. Specialized hardware for the decoding/encoding of G.729 is available from hardware manufacturers such as Sangoma and Digium (and likely others).

Asterisk does a fairly good job of supporting codecs, but is mostly focused on the codecs typically used by telephone applications (as opposed to codes used for, say, music or video such as MP3 or MP4). These are listed in Table 2-6.

3 The term *codec* is short for "coder decoder."

Table 2-6. Common codec translators

Name	Purpose
`codec_alaw`	A-law PCM codec used all over the world on the PSTN (except Canada/USA). This codec (along with ulaw) should be enabled on all your channels.
`codec_g729`	Was until recently a patented codec, but is now royalty-free. As of this writing it is still sold by Digium as an add-on, but it can also be found as a free package. It's a very popular codec if compression is desired (and CPU use is not an issue), but it imposes load on the CPU, adds latency to calls, reduces quality slightly, and will not reduce overhead in any way.
`codec_a_mu`	A-law to mu-law direct converter.
`codec_g722`	Wideband audio codec.
`codec_gsm`	Global System for Mobile Communications (GSM) codec. Very poor sound quality.
`codec_ilbc`	Internet Low Bitrate Codec.
`codec_lpc10`	Linear Predictive Coding vocoder (extremely low bandwidth).
`codec_opus`	Intended to replace speex (and vorbis).
`codec_resample`	Resamples between 8-bit and 16-bit signed linear.
`codec_speex`	Speex codec.
`codec_ulaw`	Mu-law PCM codec used on PSTN in Canada/USA. It's more formally written as μ-law, but not many people have a Greek letter μ on their keyboard, so it's popularly written as ulaw.[a] This is often the default codec, and should be enabled on all your channels.

[a] Spoken, you should say "mew-law," but again, you'll hear this pronounced "you-law" very often.

Digium distributes some additional useful codec modules: `codec_g729`, `codec_silk`, `codec_siren7`, and `codec_siren14`. These codec modules are not open source for various reasons. You must purchase a license to use `codec_g729`, but the others are free. You can find them on the Digium site.

Format Interpreters

Format interpreters (Table 2-7) perform a similar function as codec translators, but they do their work on files rather than channels, and handle more than just audio. If you have a recording on a menu that has been stored as GSM, you would need to use a format interpreter to play that recording to any channels not using the GSM codec.[4]

If you store a recording in several formats simultaneously (such as WAV, GSM, etc.), Asterisk will determine the least costly format[5] to use when a channel needs to play that recording.

4 It is partly for this reason that we do not recommend the default GSM format for system recordings. WAV recordings will sound better and use fewer CPU cycles.

5 Some codecs will impose a significant load on the CPU, so much so that a system that might support several hundred channels without transcoding will only handle a few dozen when transcoding is in use.

Table 2-7. Format interpreters

Name	Plays files stored in
`format_g729`	G.729: *.g729*
`format_gsm`	RPE-LTP (original GSM codec): *.gsm*
`format_h264`	H.264 video: *.h264*
`format_ilbc`	Internet Low Bitrate Codec: *.ilbc*
`format_jpeg`	Graphic file: *.jpeg*, *.jpg*
`format_ogg_vorbis`	Ogg container: *.ogg*
`format_pcm`	Various Pulse-Coded Modulation formats: *.alaw*, *.al*, *.alw*, *.pcm*, *.ulaw*, *.ul*, *.mu*, *.ulw*, *.g722*, *.au*
`format_siren14`	G.722.1 Annex C (14 kHz): *.siren14*
`format_siren7`	G.722.1 (7 kHz): *.siren7*
`format_sln`	8-bit signed linear: *.sln*, *.raw*
`format_vox`	*.vox*
`format_wav`	*.wav*
`format_wav_gsm`	GSM audio in a WAV container: *.wav*, *.wav49*

Dialplan Functions

Dialplan functions, listed in Table 2-8, complement the dialplan applications (see "Applications" on page 11). They provide many useful enhancements to things like string handling, time and date wrangling, and ODBC connectivity.

Table 2-8. A curated list of useful dialplan functions

Name	Purpose
`func_audiohook inherit`	Allows calls to be recorded after transfer
`func_blacklist`	Writes/reads blacklist in *astdb*
`func_callcompletion`	Gets/sets call-completion configuration parameters for the channel
`func_callerid`	Gets/sets caller ID
`func_cdr`	Gets/sets CDR variable
`func_channel`	Gets/sets channel information
`func_config`	Includes `AST_CONFIG()`; reads variables from config file
`func_curl`	Uses cURL to obtain data from a URI
`func_cut`	Slices and dices strings
`func_db`	Provides *astdb* functions
`func_devstate`	Gets state of device
`func_dialgroup`	Creates a group for simultaneous dialing
`func_dialplan`	Validates that designated target exists in dialplan
`func_env`	Includes `FILE()`, `STAT()`, and `ENV()`; performs operating system actions
`func_global`	Gets/sets global variables

Name	Purpose
func_groupcount	Gets/sets channel count for members of a group
func_hangupcause	Gets/sets hangupcause information from the channel
func_logic	Includes ISNULL(), SET(), EXISTS(), IF(), IFTIME(), and IMPORT(); performs various logical functions
func_math	Includes MATH(), INC(), and DEC(); performs mathematical functions
func_odbc	Allows dialplan integration with ODBC resources
func_rand	Returns a random number within a given range
func_realtime	Performs lookups within the Asterisk Realtime Architecture (ARA)
func_redirecting	Provides access to information about where this call was redirected from
func_shell	Performs Linux shell operations and returns results
func_sprintf	Performs string format functions similar to C function of same name
func_srv	Performs SRV lookups in the dialplan
func_strings	Includes over a dozen string manipulation functions
func_timeout	Gets/sets timeouts on channel
func_uri	Converts strings to URI-safe encoding
func_vmcount	Returns count of messages in a voicemail folder for a particular user

PBX Modules

The PBX modules are peripheral modules that provide enhanced control and configuration mechanisms. For example, pbx_config is the module that loads the traditional Asterisk dialplan. The currently available PBX modules are listed in Table 2-9.

Table 2-9. PBX modules

Name	Purpose
pbx_config	This module provides the traditional, and most popular, dialplan language for Asterisk. Without this module, Asterisk cannot read *extensions.conf*.
pbx_dundi	Performs data lookups on remote Asterisk systems.
pbx_realtime	Provides functionality related to the Asterisk Realtime Architecture.
pbx_spool	Provides outgoing spool support relating to Asterisk call files.

Resource Modules

Resource modules integrate Asterisk with external resources. This group of modules has effectively turned into a catch-all for things that do not fit in other categories. We will break them into some subgroups of modules that are related.

Configuration backends

Asterisk is configured using text files in */etc/asterisk* by default. These modules, listed in Table 2-10, offer alternative configuration methods. See Chapter 15 for detailed documentation on setting up database-backed configuration.

Table 2-10. Configuration backend modules

Name	Purpose
`res_config_curl`	Pulls configuration information using cURL
`res_config_ldap`	Pulls configuration information using LDAP
`res_config_odbc`	Pulls configuration information using ODBC

Calendar integration

Asterisk includes some integration with calendar systems. You can read and write calendar information from the dialplan. You can also have calls originated based on calendar entries. The core calendar integration is provided by the `res_calendar` module. The rest of the modules provide the ability to connect to specific types of calendar servers. Table 2-11 lists the calendar integration modules.

Table 2-11. Calendar integration modules

Name	Purpose
`res_calendar`	Enables base integration to calendaring systems
`res_calendar_caldav`	Allows features provided by `res_calendar` to connect to calendars via CalDAV
`res_calendar_exchange`	Allows features provided by `res_calendar` to connect to MS Exchange
`res_calendar_icalendar`	Allows features provided by `res_calendar` to connect to Apple/Google iCalendar

Other resource modules

Table 2-12 includes the rest of the resource modules that did not fit into one of the subgroups we defined earlier in this section.

Table 2-12. Resource modules

Name	Purpose
`res_adsi`	Provides ADSI[a]
`res_agi`	Provides the Asterisk Gateway Interface (see Chapter 18)
`res_corosync`	Provides distributed message waiting indication (MWI) and device state notifications via the Corosync Cluster Engine
`res_crypto`	Provides cryptographic capabilities
`res_curl`	Provides common subroutines for other cURL modules
`res_fax`	Provides common subroutines for other fax modules
`res_fax_spandsp`	Plug-in for fax using the `spandsp` package
`res_http_post`	Provides POST upload support for the Asterisk HTTP server
`res_http_websocket`	Provides WebSocket support for the Asterisk internal HTTP server (required by WebRTC)
`res_monitor`	Provides call-recording resources
`res_musiconhold`	Provides music on hold (MOH) resources
`res_mutestream`	Allows muting/unmuting of audio streams
`res_odbc`	Provides common subroutines for other ODBC modules

Name	Purpose
res_phoneprov	Provisions phones from Asterisk HTTP server
res_pktccops	Provides PacketCable COPS resources
res_security_log	Enables logging of security events generated by other parts of Asterisk
res_snmp	Provides system status information to an SNMP-managed network
res_speech	Generic speech recognition API[b]
res_stasis	Ties together the various components of the Stasis application infrastructure
res_xmpp	Provides XMPP resources (FKA Jabber)

[a] While most of the ADSI functionality in Asterisk is never used, the voicemail application uses this resource.
[b] Requires a separately licensed product in order to be used.

Add-on Modules

Add-on modules are community-developed modules with different usage or distribution rights from those of the main code (Table 2-13). They are kept in a separate directory and are not compiled and installed by default. To enable these modules, use the `menuselect` build configuration utility.

Table 2-13. Add-on modules

Name	Purpose	Popularity/status
chan_ooh323	Enables making and receiving VoIP calls using the H.323 protocol	Usable
format_mp3	Allows Asterisk to play MP3 files	Usable
res_config_mysql	Uses a MySQL database as a real-time configuration backend	Useful

Test Modules

Test modules are used by the Asterisk development team to validate new code. They are constantly changing and being added to, and are not useful unless you are developing Asterisk software.

If you are an Asterisk developer, however, the Asterisk Test Suite may be of interest to you, as you can build automated tests for Asterisk and submit those back to the project, which runs on several different operating systems and types of machines. By expanding the number of tests constantly, the Asterisk project avoids the creation of regressions in code. By submitting your own tests to the project, you can feel more confident in future upgrades.

More information about building tests is available in the "Asterisk Test Suite" document, or you can join the *#asterisk-testing* channel on the Freenode IRC network.

File Structure

Asterisk is a complex system, composed of many resources. These resources make use of the filesystem in several ways. Since Linux is so flexible in this regard, it is helpful to understand what data is being stored, so that you can understand where you are likely to find a particular bit of stored data (such as voicemail messages or logfiles).

Configuration Files

The Asterisk configuration files include *extensions.conf*, *pjsip.conf*, *modules.conf*, and dozens of other files that define parameters for the various channels, resources, modules, and functions that may be in use.

These files will normally be found in */etc/asterisk*. You will be working in this folder a lot as you configure and administer your Asterisk system.

Modules

Asterisk modules are usually installed to the */usr/lib/asterisk/modules* folder. You will not normally have to interact with this folder; however, it will be occasionally useful to know where the modules are located. For example, if you upgrade Asterisk and select different modules during the `menuselect` phase of the install, the old (incompatible) modules from the previous Asterisk version will not be deleted, and you will get a warning from the install script. Those old files will need to be deleted from the *modules* folder. This can be done either manually or with the "uninstall" make (`make uninstall`) target.

The Resource Library

There are several resources that require external data sources. For example, music on hold (MOH) can't happen unless you have some music to play. System prompts also need to be stored somewhere on the hard drive. The */var/lib/asterisk* folder is where system prompts, AGI scripts, music on hold, and other resource files are stored.

The Spool

The *spool* is where applications store files on a Linux system that are going to change frequently, or that will be processed by other processes at a later time. For example, Linux print jobs and pending emails are normally written to the spool until they are processed.

In Asterisk, the spool is used to store transient items such as voice messages, call recordings,[6] call files, and so forth.

The Asterisk spool will be found under the */var/spool/asterisk* directory.

Logging

Asterisk is capable of generating several different kinds of logfiles. The */var/log/asterisk* folder is where call detail records (CDRs), channel events from CEL, debug logs, queue logs, messages, errors, and other output are written.

This folder will be extremely important for any troubleshooting efforts you undertake. We will talk more about how to make use of Asterisk logs in Chapter 21.

The Dialplan

The dialplan is the heart of Asterisk. All channels that arrive in the system will be passed through the dialplan, which contains the call-flow scripts that determine how incoming calls are handled.

Dialplan is typically written using Asterisk's own dialplan syntax, which is stored in a file named */etc/asterisk/extensions.conf*. There are other ways to control call flow, and we will explore them later, but no matter which method you eventually employ, you will find that a basic understanding of the traditional dialplan will be immensely helpful. That is what we will focus on for most of the first two-thirds of this book.

Later, we will explore handling call flow outside of the dialplan, using technologies such as AMI, AGI, and ARI.

Hardware

Asterisk is capable of communicating with a vast number of different technologies. In general, these connections are made across a TCP/IP network connection (usually using SIP). However, connections to more traditional telecom circuits, such as PRI (T1, E1, etc.), BRI (EuroISDN) SS7 (mostly T1 and E1), and analog (everything from a few FXO and FXS ports up to large channel banks fed through T1/E1 CAS/RBS connections), can also be achieved using physical cards installed in a server.

Many companies produce this hardware, such as Digium (the sponsor, owner, and primary developer of Asterisk), Sangoma (who recently purchased Digium), Dialogic (also a Sangoma company), OpenVox, Pika, Voicetronix, beroNet, and many others. All of these companies have been involved with Asterisk for many years.

6 Not call detail records (CDRs), but rather audio recordings of calls generated by the `MixMonitor()` and related applications.

The most popular hardware for Asterisk is generally designed to work through the Digium Asterisk Hardware Device Interface (known as DAHDI). This is a complex architecture, and is out of the scope of this book. Server-based telephony cards will all have installation requirements unique to the manufacturer, and will require you to have strong skills in both Linux hardware installation as well as traditional PSTN circuit troubleshooting and provisioning.

If you need to interface with traditional PSTN circuits using Asterisk, we recommend that you keep Asterisk as a SIP-only platform, and interface using a third-party gateway of some sort. Be warned: this is not entry-level stuff, and if you are just starting out with Asterisk, you are strongly advised to keep your initial solutions to SIP-only.

Asterisk Versioning

The Asterisk release methodology has gone through several styles over time. This has led to some confusion in the past, but these days the versioning is fairly straightforward, and relatively easy to understand. Digium has maintained an excellent reference at the Asterisk wiki, and we encourage you to go there for the latest details on Asterisk versions.

This book was written and tested using version 16, but you will find that the fundamental concepts we explore will be relevant to most Asterisk versions. The conceptual structure of Asterisk has not changed for quite some time, and as of this writing there are no known plans to change that going forward. Future versions will deliver more powerful multimedia and conferencing capabilities, to be sure, but they are likely to be implemented within the existing structure.

Conclusion

Asterisk is composed of many different technologies, most of which are complicated in their own right. As a result, understanding Asterisk architecture can be overwhelming. Still, the reality is that Asterisk is well designed for what it does and, in our opinion, has achieved a remarkable balance between flexibility and complexity.

CHAPTER 3
Installing Asterisk

I long to accomplish great and noble tasks, but it is my chief duty to accomplish humble tasks as though they were great and noble. The world is moved along, not only by the mighty shoves of its heroes, but also by the aggregate of the tiny pushes of each honest worker.

—Helen Keller

In this chapter we're going to walk through the installation of Asterisk from the source code. Many people shy away from this method, claiming that it is too difficult and time-consuming. Our goal here is to demonstrate that installing Asterisk from source is not actually that difficult to do. More importantly, we want to provide you with the best Asterisk platform on which to learn.

In this book we will be helping you build a functioning Asterisk system from scratch. Toward that goal, in this chapter we will build a base platform for your Asterisk system. Since we are installing from source, there is potentially a lot of variation in how you can do this. Our goal here is to deliver a standard sort of platform, suitable for explorations in many areas. It is possible to strip Asterisk down to the very basics and run a very lean machine; however, that exercise is left up to the reader. The process we discuss here is designed to get you up and running quickly and simply, without short-changing you on access to interesting features.

Most of the commands you see are going to be best handled with a series of copy-paste operations (in fact, we strongly recommend you have an electronic version of this book handy for that very purpose).[1] While it looks like a lot of typing, the commands we take you through can get you from start to finish in less than 30 minutes,

1 It's been released under a Creative Commons license, so if you have purchased a hard copy (and we thank you!), you can also download a soft copy for searching and copying/pasting.

so it's really not as complex as it might appear. We run some prerequisites, some compilation, and some post-install config, and Asterisk is ready to go.

For the sake of brevity, these steps will be performed on a CentOS 7 system. This is functionally equivalent to RHEL, and similar enough to Fedora that the steps should be quite similar. For other platforms such as Debian/Ubuntu and so forth, the instructions will also be similar, but you will need to adjust as needed for your platform.[2]

The first part of the installation instructions will not deal with Asterisk as such, but rather some of the dependencies that either Asterisk requires or are necessary for some of the more useful features (such as database integration). We'll try to keep the instructions general enough that they should be useful on any distribution of your choice.

These instructions assume that you are an experienced Linux administrator.[3] A fully working Asterisk system will consist of enough discrete parts that you will find it challenging to deal with all of it if you have little or no Linux background. We'd still encourage you to dive right in, but please allow for the fact that there will be a steep learning curve if you don't already have solid Linux command-line experience.

If you want to learn the Linux command line, one of the best books we've found is *The Linux Command Line* by William Shotts, which has been released under a Creative Commons license, and dives straight into all the knowledge you need to use the Linux shell effectively. It can be found at linuxcommand.org. You could memorize the book from front to back, and pretty much everything you'd learned would be something any seasoned Linux administrator would agree was worth knowing.

Another fantastic book is of course the legendary *UNIX and Linux System Administration Handbook* by Dan Mackin, Ben Whaley, Trent R. Hein, Garth Snyder, and Evi Nemeth (Prentice Hall). Highly recommended.

2 Asterisk should run on pretty much any Linux platform, and if you are familiar with the basic process of installing software on a Linux machine, you should find Asterisk a fairly straightforward installation.

3 By which we mostly mean that you are comfortable administering a system exclusively from the shell.

Asterisk Packages

There are Asterisk packages that can be installed using package management systems such as *yum* or *apt-get*. You are encouraged to use them once you are familiar with Asterisk.

If you are using RHEL, Asterisk is available from the EPEL repository from the Fedora project. Asterisk packages are available in the Universe repository for Ubuntu.

You should also note that because of Asterisk's history, it is able to integrate with a multitude of telephony technologies; however, these days, someone new to Asterisk is going to want to learn SIP integration before worrying about more complex, obsolete or peripheral channel types. Once you are comfortable with Asterisk in a pure SIP environment, it'll be much easier to look at integrating other channel types.

Asterisk-Based Projects

Many projects use Asterisk as their underlying platform. Some of these, such as the FreePBX GUI, have become so popular that many people mistake them for the Asterisk product itself. In fact, the FreePBX GUI is so ubiquitous it is found in most of the well-known Asterisk-based projects. These projects take the base Asterisk product and add a web-based administration interface, a complex database, and external functions that are useful in a typical PBX (such as set provisioning, a time server, and so forth).

We have chosen not to cover these projects in this book, for several reasons:

- This book tries, as much as possible, to focus on Asterisk and only Asterisk.
- Books have already been written about many of these Asterisk-based projects.
- We believe that if you learn Asterisk in the way that we will teach you, the knowledge will serve you well regardless of whether or not you eventually choose to use one of these prepackaged versions of Asterisk.
- If you want to be able to make sense of what's going on under the hood of a FreePBX-based system, this book will introduce you to some of the skills you will need.
- For us, the power of Asterisk is that it does not attempt to solve your problems for you. These projects are truly amazing examples of what can be built with Asterisk. However, if you are looking to build your own Asterisk application (which is really what Asterisk is all about), these projects might create needless obstacles, simply because they are focused on simplifying the process of building a business PBX, not on making it possible to access the full potential of the Asterisk platform.

Some of the most popular Asterisk-based projects include (in no particular order):

AsteriskNOW
: Managed by Digium. Uses FreePBX GUI.

Issabel
: A fork of the original open source releases of the Elastix product.[4] Uses FreePBX GUI.

The Official FreePBX Distro
: The official distro of the FreePBX project. Managed by Sangoma.

Asterisk for Raspberry Pi
: A complete install of Asterisk and FreePBX for the Raspberry Pi.

AstLinux
: The AstLinux project caters to a community that want to run Asterisk on small, low-power, solid-state devices. The install size of the entire solution is measured in megabytes (AstLinux was originally designed to fit on CompactFlash cards). If you are fascinated by small computers, and want to play with a PBX-in-a-box that fits in your pocket, AstLinux may be for you.

We recommend that you check them out.[5]

Linux Installation

Asterisk is developed using Linux, and unless you're very comfortable with porting software between various platforms, that is what you're going to want to use.

In this book, we're going to use CentOS as the platform. If you would prefer a different Linux distro, it is expected that you have sufficient Linux skills to understand what some of the differences may be. These days, it's so easy and cheap to fire up an instance of any common distribution that there's no real harm in using CentOS to learn, and then migrate to whatever you prefer when you're ready.

We recommend installing the Minimal version of CentOS, since the installation process we will be going through handles all the prerequisites. This also ensures you're not installing anything you don't need.

4 Elastix is no longer an Asterisk-based or open source product.

5 After you read our book, of course.

Choosing Your Platform

OK, so strictly speaking we've already chosen your platform for you, but there are several different ways to get a CentOS server up and running (see Table 3-1).

Table 3-1. Comparing Linux platforms that are suitable for Asterisk

Platform	Pros	Cons
OpenStack (DigitalOcean, Linode, VULTR, etc.)	Up and running in minutes. Inexpensive to operate. Doesn't require any resources on your local system. Accessible from anywhere. Can be used in a production environment. Fantastic for quick prototyping projects.	You pay as long as it's running. The IP address is only yours for as long as the system is running. Requires some DevOps skills if you want to deploy in production. No firewall in place by default.
VirtualBox (or other PC-based platform)	Free to use. No external exposure. Excellent for small lab projects.	Requires more horsepower on your system. Requires storage space on your local system. Not easy to deploy into a production environment.
AWS and/or Lightsail	Inexpensive to operate. Doesn't require any resources on your local system. Accessible from anywhere. Can be used in a production environment. Scales to enormous sizes.	You pay as long as it's running. Somewhat more skills required to gather all the resources you need.
Physical hardware	Dedicated platform. Can be shipped and installed anywhere. Complete control over all aspects of environment, hardware, network, and so forth.	Risk of component failure. Power consumption. Noise. Potential costs for hosting. No inherent redundancy.
Other (really anything that'll run CentOS 7 should be fine)	You can use an environment that you're familiar with.	You're on your own.
Other Linux (you don't actually have to run CentOS)	You can run the exact environment you want.	You need to have strong Linux admin skills.

For the purposes of learning, we recommend one of two simple ways to get going:

- *If you are running Windows as your desktop*: Download VirtualBox, then download the CentOS 7 Minimal ISO, and install on your local machine.
- *If you are comfortable working with SSH-based, keyed connections to remote systems*: Create a hosted system (for example, a DigitalOcean CentOS droplet).

This book was developed and tested using both VirtualBox and DigitalOcean.

VirtualBox Steps

Grab a copy of VirtualBox from the platform's website and install it.

Download the Minimal ISO from the Centos website.

Get yourself a copy of PuTTY if you're using Windows.

Create a new virtual machine with the following characteristics:

- Type: Linux
- Version: Red Hat (64-bit)
- Memory size: 2048 MB
- Hard disk: Create a virtual hard disk now
- File location: Pick a good spot for storing your virtual machine images
- File size: 16 GB is fine for what we're doing here, but something larger would be needed for production

Once the basic machine has been defined, you'll need to tweak it as follows:

- Storage: Under **Storage**, **Controller: IDE ...**

  1. You should see the CD/DVD has a tiny disc icon labeled **Empty**.
  2. Click on it and to the right under **Attributes**, there'll be another tiny disc icon.
  3. Click on that, and it'll ask you to **Choose Optical Virtual Disk File**.
  4. Locate on your hard drive the Minimal ISO you downloaded from CentOS, and choose it.
  5. The Storage Tree should now show the CentOS ISO.

- Network: Adaptor 1

  Attached to: **Bridged Adapter**

Start up the machine you've just created, and it should take you through a basic installation of CentOS. Here are a few items you'll want to specify (for anything else, the defaults should do):

- Date and time: Adjust to your time zone if you wish.
- Network and host name: **Ethernet**—toggle from off to **on** (it should immediately grab an IP address from your network; if not, set one manually). Press the **Done** button.
- Installation destination: It may require you to confirm the target, but you shouldn't need to change anything. Press the **Done** button.
- That's it. **Begin Installation**.

While the installation is taking place, set the root password, and also create a user named `astmin`. Make the `astmin` user an administrator.

The installation will take a few minutes. Grab a coffee!

Once the install is done, the installer will ask you to **Reboot**. The reboot should only take 15 seconds or so.

Congratulations, your system is ready. Log in as `root`.

Linux (OpenStack) Host

You'll obviously need an account with a hosted Linux provider if you're going to use this method (we've found OpenStack-based offerings to be the cheapest, relative to the quality/performance/simplicity offered). We've been using DigitalOcean for many years, but have also found Linode and VULTR to be strong providers in this space.[6] Once you've got that sorted, you can log in and create a new system something like the following:

- CentOS 7 (lastest version, 64-bit)
- 4 GB 2vCPUs (we don't really need the 4 GB RAM, but it is good to have the 2xCPUs; you can probably get away with 2 GB 1vCPU, if you're really cost-conscious)
- Data center closest to you

Once that's up and running, log in as the default user (as of this writing, it's `centos`).

Note that DigitalOcean instances do not have a firewall by default. Instead, they provide a firewall as a part of their environment. The system you build will therefore not have any native firewall in place, and will be subject to external attacks shortly after you complete configuration (you'll see this on the Asterisk console). Different providers will have different firewall policies. You are responsible for making sure your firewalling is working correctly. We'll be discussing security and anti-fraud in more detail later on in this book.

Dependencies

The system you've just built isn't really much more than a basic bootstrapped system. In order to prepare it for an Asterisk installation, there are a few things we'll need to install first.

6 Amazon's new Lightsail service also promises to simplify the creation of hosted Linux machines.

The following commands can be typed from the command line, or added to a simple shell script and run that way.

```
sudo yum -y update &&
sudo yum -y install epel-release &&
sudo yum -y install python-pip &&
sudo yum -y install vim wget dnf&&
sudo pip install alembic ansible &&
sudo pip install --upgrade pip &&
sudo mkdir /etc/ansible &&
sudo chown astmin:astmin /etc/ansible &&
sudo echo "[starfish]" >> /etc/ansible/hosts &&
sudo echo "localhost ansible_connection=local" >> /etc/ansible/hosts &&
mkdir -p ~/ansible/playbooks
```

We've installed Ansible simply because it's a quick and easy way to get all the dependencies met. We've written a playbook to perform the following operations:

1. Install `dnf`, `vim`, `wget`, and `MySQL-python`.
2. Install the MySQL community-edition repository.
3. Install `mysql-server`.
4. Tweak some variables in the `mysql-server` installation.
5. Start the mysql-server daemon.
6. Modify some MySQL credentials (create users, set passwords).
7. Create a MySQL database/schema for Asterisk to use.
8. Apply some security best practices (remove anonymous user, test database, etc.).
9. Create `asterisk` user.
10. Create `astmin` user.
11. Install dependencies for ODBC.
12. Install some diagnostic tools.
13. Tweak the firewall to allow SIP and RTP traffic.
14. Edit some ODBC config files.

This can all be done manually, but it's just a lot of typing, and Ansible is really good at streamlining this process.

Create an Ansible playbook in the file *~/ansible/playbooks/starfish.yml*.

The *libmyodbc8a.so* file is versioned, so, if you don't have version 8 of UnixODBC:

1. Run the playbook the first time (to install the UnixODBC library).
2. Find out what file was installed at */usr/lib64/libmyodbc<version>a.so*.
3. Edit the playbook as appropriate (provide the correct filename).
4. Save and rerun the playbook (which will then update the configuration files to point to the correct library).

Here's the playbook:

```
---
- hosts: starfish
  become: yes
  vars:
# Use these on the first run of this playbook
    current_mysql_root_password: ""
    updated_mysql_root_password: "YouNeedAReallyGoodPassword"
    current_mysql_asterisk_password: ""
    updated_mysql_asterisk_password: "YouNeedAReallyGoodPasswordHereToo"
# Comment the above out after the first run

# Uncomment these for subsequent runs
#    current_mysql_root_password: "YouNeedAReallyGoodPassword"
#    updated_mysql_root_password: "{{ current_mysql_root_password }}"
#    current_mysql_asterisk_password: "YouNeedAReallyGoodPasswordHereToo"
#    updated_mysql_asterisk_password: "{{ current_mysql_asterisk_password }}"

  tasks:

  - name: Install epel-release
    dnf:
      name: epel-release
      state: present

  - name: Install dependencies
    dnf:
      name: ['vim', 'wget', 'MySQL-python']
      state: present

  - name: Install the MySQL repo.
    dnf:
      name: http://repo.mysql.com/mysql-community-release-el7-5.noarch.rpm
      state: present

  - name: Install mysql-server
    dnf:
      name: mysql-server
      state: present

  - name: Override variables for MySQL (RedHat).
```

```
    set_fact:
      mysql_daemon: mysqld
      mysql_packages: ['mysql-server']
      mysql_log_error: /var/log/mysqld.err
      mysql_syslog_tag: mysqld
      mysql_pid_file: /var/run/mysqld/mysqld.pid
      mysql_socket: /var/lib/mysql/mysql.sock
    when: ansible_os_family == "RedHat"

  - name: Ensure MySQL server is running
    service:
      name: mysqld
      state: started
      enabled: yes

  - name: update mysql root pass for localhost root account from local servers
    mysql_user:
      login_user: root
      login_password: "{{ current_mysql_root_password }}"
      name: root
      host: "{{ item }}"
      password: "{{ updated_mysql_root_password }}"
    with_items:
        - localhost

  - name: update mysql root password for all other local root accounts
    mysql_user:
      login_user: root
      login_password: "{{ updated_mysql_root_password }}"
      name: root
      host: "{{ item }}"
      password: "{{ updated_mysql_root_password }}"
    with_items:
        - "{{ inventory_hostname }}"
        - 127.0.0.1
        - ::1
        - localhost.localdomain

  - name: create asterisk database
    mysql_db:
      login_user: root
      login_password: "{{ updated_mysql_root_password }}"
      name: asterisk
      state: present

  - name: asterisk mysql user
    mysql_user:
      login_user: root
      login_password: "{{ updated_mysql_root_password }}"
      name: asterisk
      host: "{{ item }}"
      password: "{{ updated_mysql_asterisk_password }}"
      priv: "asterisk.*:ALL"
    with_items:
        - "{{ inventory_hostname }}"
        - 127.0.0.1
        - ::1
        - localhost
        - localhost.localdomain
```

```
  - name: remove anonymous user
    mysql_user:
      login_user: root
      login_password: "{{ updated_mysql_root_password }}"
      name: ""
      state: absent
      host: "{{ item }}"
    with_items:
        - localhost
        - "{{ inventory_hostname }}"
        - 127.0.0.1
        - ::1
        - localhost.localdomain

  - name: remove test database
    mysql_db:
      login_user: root
      login_password: "{{ updated_mysql_root_password }}"
      name: test
      state: absent

  - user:
      name: asterisk
      state: present
      createhome: yes

  - group:
      name: asterisk
      state: present

  - user:
      name: astmin
      groups: asterisk,wheel
      state: present

  - name: Install other dependencies
    dnf:
      name:
      - unixODBC
      - unixODBC-devel
      - mysql-connector-odbc
      - MySQL-python
      - tcpdump
      - ntp
      - ntpdate
      - jansson
      - bind-utils
    state: present

#   Tweak the firewall for UDP/SIP
  - firewalld:
      port: 5060/udp
      permanent: true
      state: enabled

#   Tweak firewall for UDP/RTP
  - firewalld:
      port: 10000-20000/udp
```

```
      permanent: true
      state: enabled

  - name: Ensure NTP is running
    service:
      name: ntpd
      state: started
      enabled: yes

# The libmyodbc8a.so file is versioned, so if you don't have version 8, see what the
# /usr/lib64/libmyodbc<version>a.so file is, and refer to that instead
# on your 'Driver64' line, and then run the playbook again
  - name: update odbcinst.ini
    lineinfile:
      dest: /etc/odbcinst.ini
      regexp: "{{ item.regexp }}"
      line: "{{ item.line }}"
      state: present
    with_items:
      - regexp: "^Driver64"
        line: "Driver64 = /usr/lib64/libmyodbc8a.so"
      - regexp: "^Setup64"
        line: "Setup64 = /usr/lib64/libodbcmyS.so"

  - name: create odbc.ini
    blockinfile:
      path: /etc/odbc.ini
      create: yes
      block: |
        [asterisk]
        Driver = MySQL
        Description = MySQL connection to 'asterisk' database
        Server = localhost
        Port = 3306
        Database = asterisk
        UserName = asterisk
        Password = {{ updated_mysql_asterisk_password }}
        #Socket = /var/run/mysqld/mysqld.sock
        Socket = /var/lib/mysql/mysql.sock
...
```

Run the playbook with the following command:

```
$ ansible-playbook ~/ansible/playbooks/starfish.yml
```

Sit back and watch the magic happen.

Once Ansible has completed the assigned tasks, verify that ODBC can connect to the database using the `asterisk` user credentials.

```
$ echo "select 1" | isql -v asterisk asterisk password
```

You should see a result something like this:

```
+---------------------------------------+
| Connected!                            |
| sql-statement                         |
| help [tablename]                      |
| quit                                  |
+---------------------------------------+
```

```
SQL> select 1
+--------------------+
| 1                  |
+--------------------+
| 1                  |
+--------------------+
SQLRowCount returns 1
1 rows fetched
```

If you do not see the `Connected!` message, you need to troubleshoot your database and ODBC installation. The first thing you should do is make sure you can log into the database from the command line using the `asterisk` user (`mysql -u asterisk -p`). Most ODBC problems tend to end up being credentials problems (i.e., wrong password or username), so work backward to ensure all the credentials work as they should, and double-check that you didn't get any problem messages from Ansible.

As of this writing, the version of *jansson* installed from the EPEL repo is an older version than the one Asterisk requires, so we'll have to install that manually.

The system is now prepared, and we're ready to download and install Asterisk.

Asterisk Installation

Asterisk is officially delivered in a tarball (as source code), and it must be downloaded, extracted, and compiled.[7] This is not difficult to do, so long as you have all the dependencies correct. Between the writing of this book and your reading of it, there may have been some changes to the various dependencies, so your install process may have to be run slightly differently. It's often difficult to know the difference between an error message that can safely be ignored, and one that is indicating a critical problem; however, in general, you should have identified and resolved any errors in the previous processes before arriving at this step. If your dependencies are sorted, the Asterisk install will tend to go smoothly.

Download and Prerequisites

Log out of the system, and log back in as user `astmin`.[8]

Type the following commands from the shell in order to download the Asterisk source code:

7 Note that members of the community will also produce packaged versions of Asterisk. The EPEL repository, for example, maintains a version that can be installed using `dnf` (`yum`). As of this writing, only the tarball version is officially maintained, and we recommend this method at this time, mostly due to the many different modules that come with Asterisk, and the usefulness in being able to build what you need from source.

8 On a DigitalOcean instance, you'll need to ensure your SSH key is in the file */home/astmin/.ssh/authorized_keys*.

When you see us write <TAB> in a filename, what we mean is that you should press the Tab key on your keyboard and allow auto-complete to fill in what it can. The rest of the typing then follows.

```
$ mkdir ~/src
$ cd ~/src
$ wget https://downloads.asterisk.org/pub/telephony/asterisk/asterisk-16-current.tar.gz
$ tar zxvf asterisk-16-current.tar.gz
$ cd asterisk-16.<TAB>      # tab should auto-complete (unless it has more than one match)
```

We can now run a few prerequisites that the Asterisk team has defined, and also have the environment checked:

```
$ cd contrib/scripts (or cd ~/src/asterisk-16.<TAB>/contrib/scripts
$ sudo ./install_prereq install    # asterisk has a few prerequisites that this simplifies
$ cd ../..
$ ./configure --with-jansson-bundled
```

Asterisk is now ready to compile and install, but there are a few tweaks worth making to the configuration before compilation.

Compiling and Installing

```
$ make menuselect
```

You will see a menu that presents various options you can select for the compiler. Use the arrow and Tab keys to move around, and the Enter key to select/deselect. For the most part, the defaults should be fine, but we want to make a few tweaks to the sound files in order to ensure we have all the sounds we want, in the best format.

At this point you can also select other languages you wish to have on your system. We recommend you select the WAV and G722 formats (and G729 as well, if you need to support it).

Under Codec Translators (`--- External ---`):

- Select `[*] codec_opus`
- Select `[*] codec_silk`
- Select `[*] codec_siren7`
- Select `[*] codec_siren14`
- Select `[*] codec_g729a`

Under Core Sound Packages:

- Deselect `[*] CORE-SOUNDS-EN-GSM`
- Select `[*] CORE-SOUNDS-EN-WAV`
- Select `[*] CORE-SOUNDS-EN-G722`

Under Extras Sound Packages:

- Select `[*] EXTRA-SOUNDS-EN-WAV`
- Select `[*] EXTRA-SOUNDS-EN-G722`

Save and Exit.

Three more commands and Asterisk is installed:

```
$ make               # this will take several minutes to complete
                     # (depending on the speed of your system)
$ sudo make install  # you must run this with escalated privileges
$ sudo make config   # this too
```

When the `make config` command has completed, it will suggest some commands to install the sample configuration files. For the purposes of this book, *you **do not** want to do this*. We will be building the necessary files by hand, so the sample files will only serve to disrupt and confuse that process. Having said that, the sample files are useful, and we will mention them throughout this book, since they are excellent reference material.

Reboot the system.

Once the boot is complete, log back in as the `astmin` user, and temporarily set SELinux to `Permissive` (it will revert to `Enforcing` after each boot, so until we've sorted out the SELinux portion of the install, this has to happen on every boot):

```
$ sudo setenforce  Permissive

$ sudo sestatus
```

This should show `Current mode: permissive`

Verify that Asterisk is running with the following command:

```
$ ps -ef | grep asterisk
```

You want to see the `/user/sbin/asterisk` daemon running (currently as user `root`, but we'll fix that shortly).

Asterisk is now installed and is running; however, there are a few configuration settings we'll need to make before the system is in any way useful.

Initial Configuration

Asterisk stores its configuration files in the */etc/asterisk* folder by default. The Asterisk process itself doesn't need any configuration files in order to run; however, it will not be usable yet, since none of the features it provides have been specified. We're going to handle a few of the initial configuration tasks now.

Asterisk configuration files use the semicolon (;) character for comments, primarily because the hash character (#) is a valid character on a telephone number pad.

The *modules.conf* file gives you fine-grained control over what modules Asterisk will (and will not) load. It's usually not necessary to explicitly define each module in this file, but you could if you wanted to. We're going to create a very simple file like this:

```
$ sudo chown asterisk:asterisk /etc/asterisk ; sudo chmod 664 /etc/asterisk

$ sudo -u asterisk vim /etc/asterisk/modules.conf

[modules]
autoload=yes
preload=res_odbc.so
preload=res_config_odbc.so
```

We're using ODBC to load many of the configurations of other modules, and we need this connector available before Asterisk attempts to load anything else, so we'll preload it.

Next up, we're going to tweak the *logger.conf* file just a bit from the defaults.

```
$ sudo -u asterisk vim /etc/asterisk/logger.conf

[general]
exec_after_rotate=gzip -9 ${filename}.2;
[logfiles]
;debug => debug
;security => security
console => notice,warning,error,verbose
;console => notice,warning,error,debug
messages => notice,warning,error
full => notice,warning,error,debug,verbose,dtmf,fax
;full-json => [json]debug,verbose,notice,warning,error,dtmf,fax
;syslog keyword : This special keyword logs to syslog facility
;syslog.local0 => notice,warning,error
```

You will notice that many lines are commented out. They're there as a reference, because you'll find when debugging your system you may want to frequently tweak this file. We've found it's easier to have a few handy lines prepared and commented out, rather than having to look up the syntax each time.

The next file, *asterisk.conf*, defines various folders needed for normal operation, as well as parameters needed to run as the `asterisk` user:

```
$ sudo -u asterisk vim /etc/asterisk/asterisk.conf

[directories](!)
astetcdir => /etc/asterisk
astmoddir => /usr/lib/asterisk/modules
astvarlibdir => /var/lib/asterisk
astdbdir => /var/lib/asterisk
astkeydir => /var/lib/asterisk
astdatadir => /var/lib/asterisk
astagidir => /var/lib/asterisk/agi-bin
astspooldir => /var/spool/asterisk
[options]
astrundir => /var/run/asterisk
astlogdir => /var/log/asterisk
astsbindir => /usr/sbin
runuser = asterisk    ; The user to run as. The default is root.
rungroup = asterisk   ; The group to run as. The default is root
```

We'll configure more files later on, but these are all we need for the time being.

Let's update the ownership of the files so the `asterisk` user has proper access to them.

```
$ sudo chown -R asterisk:asterisk {/etc,/var/lib,/var/spool,/var/log,/var/run}/asterisk
```

We also may need to add a rule to the */etc/tmpfiles.d* folder, to allow Asterisk to create a socket at runtime.

```
$ sudo vim /etc/tmpfiles.d/asterisk.conf

d /var/run/asterisk 0775 asterisk asterisk
```

(See `man tmpfiles.d` for more information.)

Next up, we're going to initialize the database with the tables Asterisk needs for ODBC-based configuration.

The Asterisk source files include a contribution that the Digium folks maintain as part of Asterisk, in order to version-control the database tables needed. This greatly simplifies keeping the database correct through the upgrade process.

Navigate to the relevant directory and make a copy of the configuration file.

```
$ cd ~/src/asterisk-16.<TAB>/contrib/ast-db-manage

$ cp config.ini.sample config.ini
```

Now, we're going to open the file and give it the credentials for our database (which are defined in the Ansible playbook named *starfish.yml*, under the variable `current_mysql_asterisk_password`, which we used at the beginning of this chapter):

```
$ vim config.ini
```

Find the lines that look similar to this:

```
#sqlalchemy.url = postgresql://user:pass@localhost/asterisk
sqlalchemy.url = mysql://user:pass@localhost/asterisk

# Logging configuration
[loggers]
keys = root,sqlalchemy,alembic
```

Make a copy of it, comment it out, and edit it with the correct credentials:

```
#sqlalchemy.url = postgresql://user:pass@localhost/asterisk
#sqlalchemy.url = mysql://user:pass@localhost/asterisk
sqlalchemy.url = mysql://asterisk:YouNeedAReallyGoodPasswordHereToo@localhost/asterisk

# Logging configuration
[loggers]
keys = root,sqlalchemy,alembic
```

Now, with that very simple bit of configuration, we can use Alembic to automagically configure the database perfectly for Asterisk (this used to be somewhat painful to do in past versions of Asterisk):

```
$ alembic -c ./config.ini upgrade head
```

Alembic is not used by Asterisk, so the configuration you've just performed does not allow Asterisk to use the database. All it does is run a script that creates the schema and tables Asterisk will use (you could do this manually as well, but trust us, you want Alembic to handle this). It's part of the install/upgrade process. *It's especially useful if you have live tables, with real data in them, and want to be able to upgrade and retain the relevant configuration.*

Log into the database now, and review all the tables that have been created:

```
$ mysql -u asterisk -p

mysql> use asterisk;

mysql> show tables;
```

You should see a list similar to this:

```
| alembic_version_config      |
| extensions                  |
| iaxfriends                  |
| meetme                      |
| musiconhold                 |
| ps_aors                     |
| ps_asterisk_publications    |
| ps_auths                    |
| ps_contacts                 |
| ps_domain_aliases           |
| ps_endpoint_id_ips          |
| ps_endpoints                |
| ps_globals                  |
| ps_inbound_publications     |
| ps_outbound_publishes       |
```

```
| ps_registrations           |
| ps_resource_list           |
| ps_subscription_persistence |
| ps_systems                 |
| ps_transports              |
| queue_members              |
| queue_rules                |
| queues                     |
| sippeers                   |
| voicemail                  |
```

We're not going to configure anything in the database as of yet. We've got some more base configuration to do first.

Exit the database CLI.

Now that we've got the database structure to handle Asterisk config, we're going to tell Asterisk how to connect to the database.

```
$ sudo -u asterisk vim /etc/asterisk/res_odbc.conf
```

Once again, you'll need the credentials you defined in your Ansible playbook.

```
[asterisk]
enabled => yes
dsn => asterisk
username => asterisk
password => YouNeedAReallyGoodPasswordHereToo
pre-connect => yes
```

SELinux Tweaks

We're going to install some SELinux tools, and make a few changes to the SELinux configuration so that the system will boot properly.

A common approach is to simply edit */etc/selinux/config*, and set `enforcing=disabled`. We do not recommend this, but doing so completely disables SELinux and renders the following steps unnecessary.

```
$ sudo dnf -y install setools setroubleshoot setroubleshoot-server

$ sudo vim /etc/selinux/config
```

You're going to change the line `SELINUX=enforcing` to `SELINUX=permissive`. This will ensure the logfiles show potential SELinux errors, without actually blocking the relevant processes.

Next, we're going to give Asterisk ownership of the */etc/odbc.ini* file.

```
$ sudo chown asterisk:asterisk /etc/odbc.ini

$ sudo semanage fcontext -a -t asterisk_etc_t /etc/odbc.ini
```

```
$ sudo restorecon -v /etc/odbc.ini

$ sudo ls -Z  /etc/odbc.ini
```

If all is well, you should see now that the file context for this file has been set to `asterisk_etc_t`:

```
-rw-r--r--. asterisk asterisk system_u:object_r:asterisk_etc_t:s0 /etc/odbc.ini
```

There are a few more SELinux errors we've seen here during the writing of the book. They may have been corrected by the time you read this, but there should be no harm in running them:

```
$ sudo /sbin/restorecon -vr /var/lib/asterisk/*

$ sudo /sbin/restorecon -vr /etc/asterisk*
```

Reboot the system, and we're going to check the log for any nasty SELinux errors before we set it to `enforcing`.

```
$ sudo grep -i sealert /var/log/messages
```

There may be a few messages in there complaining about things Asterisk doesn't need (for example, a hidden file named *.odbc.ini*), but so long as it's not full of errors about all sorts of important Asterisk components, you should be good to go. One last thing you have to change is an SELinux Boolean to allow Asterisk to create a TTY.

```
$ sudo setsebool -P daemons_use_tty 1
```

Edit the */etc/selinux/config* file again, this time setting `SELINUX=enforcing`. Save and reboot once more.

Verify that Asterisk is running (as user `asterisk`).

```
$ ps -ef | grep asterisk

asterisk  3992  3985  0 16:38 ?        00:00:01 /usr/sbin/asterisk -f -vvvg -c
```

OK, we're nearly done with the installation now.

Firewall Tweaks

We'll make a couple of firewall tweaks to prepare our system for SIP (and SIP Secure).

```
$ sudo firewall-cmd --zone=public --add-service=sip --permanent

$ sudo firewall-cmd --zone=public --add-service=sips --permanent
```

Final Tweaks

Your Asterisk system is ready to roll.

Let's put some initial data into the configuration files, so that in the next chapter we can begin to work with our new Asterisk system.

Since we're going to use the PJSIP channel for all of our calling, we're going to tell Asterisk to look for PJSIP configuration in the database:

```
$ sudo -u asterisk vim /etc/asterisk/sorcery.conf

[res_pjsip] ; Realtime PJSIP configuration wizard
; eventually more modules will use sorcery to connect to the
; database, but currently only PJSIP uses this
endpoint=realtime,ps_endpoints
auth=realtime,ps_auths
aor=realtime,ps_aors
domain_alias=realtime,ps_domain_aliases
contact=realtime,ps_contacts

[res_pjsip_endpoint_identifier_ip]
identify=realtime,ps_endpoint_id_ips

$ sudo -u asterisk vim /etc/asterisk/extconfig.conf

[settings] ; older mechanism for connecting all other modules to the database
ps_endpoints => odbc,asterisk
ps_auths => odbc,asterisk
ps_aors => odbc,asterisk
ps_domain_aliases => odbc,asterisk
ps_endpoint_id_ips => odbc,asterisk
ps_contacts => odbc,asterisk

$ sudo -u asterisk vim /etc/asterisk/modules.conf

[modules]
autoload=yes
preload=res_odbc.so
preload=res_config_odbc.so
noload=chan_sip.so ; deprecated SIP module from days gone by
```

We now have to place one bit of config into the *pjsip.conf* file, which defines the transport mechanism.

```
$ sudo -u asterisk vim /etc/asterisk/pjsip.conf

[transport-udp]
type=transport
protocol=udp
bind=0.0.0.0
```

Finally, let's log into the database, and define some sample configurations for PJSIP:

```
$ mysql -D asterisk -u asterisk -p

mysql>

insert into asterisk.ps_aors (id, max_contacts) values ('0000f30A0A01', 1);
insert into asterisk.ps_aors (id, max_contacts) values ('0000f30B0B02', 1);
insert
  into asterisk.ps_auths
  (id, auth_type, password, username)
  values
  ('0000f30A0A01', 'userpass', 'not very secure', '0000f30A0A01');
insert
```

```
  into asterisk.ps_auths
  (id, auth_type, password, username)
  values
  ('0000f30B0B02', 'userpass', 'hardly to be trusted', '0000f30B0B02');
insert
  into asterisk.ps_endpoints
  (id, transport, aors, auth, context, disallow, allow, direct_media)
  values
  ('0000f30A0A01', 'transport-udp', '0000f30A0A01', '0000f30A0A01',
  'sets', 'all', 'ulaw', 'no');
insert
  into asterisk.ps_endpoints
  (id, transport, aors, auth, context, disallow, allow, direct_media)
  values
  ('0000f30B0B02', 'transport-udp', '0000f30B0B02', '0000f30B0B02',
  'sets', 'all', 'ulaw', 'no');
exit
```

Let's reboot, and then we'll log into our new Asterisk system and have a look at what we've created.

Validating Your New Asterisk System

We don't need to dive too deeply into the system at this point, since all the chapters that follow will be doing exactly that.

So all we need to do is verify that we can log into the system and that the PJSIP endpoints we've created are there.

```
$ sudo asterisk -rvvvv

*CLI> pjsip show endpoints
```

You should see the two endpoints we created listed as follows:

```
 Endpoint:  <Endpoint/CID.................................>  <State.....>  <Channels.>
    I/OAuth:  <AuthId/UserName.....................................................>
        Aor:  <Aor.........................................>  <MaxContact>
      Contact:  <Aor/ContactUri..........................> <Hash....> <Status> <RTT(ms)..>
  Transport:  <TransportId........>  <Type>  <cos>  <tos>  <BindAddress.................>
   Identify:  <Identify/Endpoint.....................................................>
        Match:  <criteria.........................>
    Channel:  <ChannelId.....................................>  <State.....>  <Time.....>
        Exten: <DialedExten...........>  CLCID: <ConnectedLineCID.......>
==========================================================================================
 Endpoint:  0000f30A0A01                                        Not in use    0 of inf
     InAuth:  1/0000f30A0A01
  Transport:  transport-udp             udp      0      0  0.0.0.0:5060

 Endpoint:  0000f30B0B02                                        Unavailable   0 of inf
     InAuth:  2/0000f30B0B02
  Transport:  transport-udp             udp      0      0  0.0.0.0:5060
Objects found: 2
```

If you don't see the two endpoints listed, you've got a configuration issue. You're going to have to work backward to ensure you don't have any errors that prevent Asterisk from connecting to the database and instantiating these two endpoints.

Common Installation Errors

The following conditions (in no particular order) cause the majority of installation errors:

Syntax errors
: In some cases, substituting a tab for a space can be enough to break something. UnixODBC, for example, has proven to be sensitive to missing spaces between `key = value` definitions. The best advice we can give here is to use copy/paste whenever possible, as opposed to manual input.

Permissions problems
: These can be annoying to resolve, but error messages will generally provide the clues you need. The */var/log/messages* file is often a gold mine for useful clues.

Missing steps
: A missed step might not have any noticeable effects until many steps later. Double-check everything, and verify functionality before moving on.

Credentials problems
: Always verify that the users and passwords you create work manually, before using them in a configuration file.

It's not possible nor necessary to dig into every warning and error message you might see, but if we've provided a test to run, and it doesn't produce anything like we said it should, you should probably work through that step again until you've figured out what's going on.

Some Final Configuration Notes

Once installed, Asterisk will have created an environment for itself in your Linux machine. The following sections have some useful tidbits of information about how you can interact with your new Asterisk installation.

Sample Configuration Files for Future Reference

Even though we warned you not to run the `sudo make samples` command during the installation (because that will fill your */etc/asterisk* directory with a bunch of stuff you don't want), the sample files are nevertheless a fantastic reference. In your Asterisk source directory, you will find the following two directories:

```
~/src/asterisk-16.<TAB>/configs/basic-pbx
~/src/asterisk-16.<TAB>/configs/samples
```

The files in those folders are worth reading through (especially for any module you're working with and want to research how to do something).

Give them a read when you have a chance.

Running `make samples` on a system that already has configuration files will overwrite the existing files.

The Asterisk Shell Command

Asterisk can be run either as a daemon or as an application. In general, you will want to run it as an application when you are building, testing, and troubleshooting, and as a daemon when you put it into production.

The command to start Asterisk is the same regardless of whether you're running it as a daemon or an application:

```
asterisk
```

However, without any arguments, this command will assume certain defaults and start Asterisk as a background application. In other words, you never want to run the command `asterisk` on its own, but rather will want to pass some options to it to better define the behavior you are looking for. The following list provides some examples of common usages:

`-h`
: This command displays a helpful list of the options you can use. For a complete list of all the options and their descriptions, run the command **`man asterisk`**.

`-c`
: This option starts Asterisk as an application (in the foreground). This means that Asterisk is tied to your user session. In other words, if you close your user session by logging out or losing the connection, Asterisk dies. This is the option you will typically use when building, testing, and debugging, but you would not want to use it in production. If you started Asterisk in this manner, type **`core stop now`** at the CLI prompt to stop Asterisk and exit.

`-v`, `-vv`, `-vvv`, `-vvvv`, *etc.*
: This option can be used with other options (e.g., `-cvvv`) in order to increase the verbosity of the console output. It does exactly the same thing as the CLI command `core set verbose` *n* where *n* is any integer between 0 and 5 (any integer greater than 5 will work, but will not provide any more verbosity). Sometimes it's

useful to not set the verbosity at all. For example, if you are looking to see only startup errors, notices, and warnings, leaving verbosity off will prevent all the other startup messages from being displayed.

`-d`, `-dd`, `-ddd`, `-dddd`, *etc.*
: This option can be used in the same way as `-v`, but instead of normal output, this will specify the level of debug output (which is primarily useful for developers who wish to troubleshoot problems with the code). You will also need to enable output of debugging information in the *logger.conf* file (which we will cover in more detail in Chapter 21).

`-r`
: This command is essential if you want to connect to the CLI of an Asterisk process running as a daemon. You will probably use this option more than any other for Asterisk systems that are in production. This option will only work if you have a daemonized instance of Asterisk already running. To exit the CLI when this option has been used, type **`exit`**.

`-T`
: This option will add a timestamp to CLI output.

`-x`
: This command allows you to pass a string to Asterisk that will be executed as if it had been typed at the CLI. As an example, to get a quick listing of all the channels in use without having to start the Asterisk console, simply type **`asterisk -rx 'core show channels'`** from the shell, and you'll get the output you are looking for.

`-g`
: This option instructs Asterisk to dump a core file if it crashes.

We recommend you try out a few combinations of these commands to see what they do.

safe_asterisk

When you install Asterisk using the `make config` directive, it will create a script called *safe_asterisk*, which is run during the `init` process of Linux each time you boot.

The *safe_asterisk* script provides the following benefits:

- Restarts Asterisk automatically after a crash
- Can be configured to email the administrator if a crash has occurred
- Defines where crash files are stored (*/tmp* by default)

- Executes a script if a crash has occurred

You don't need to know too much about this script, other than to understand that it should normally be running. In most environments this script works fine in its default format.

Conclusion

In this chapter we've provided a curated example of how an Asterisk installation should go. We've chosen the Linux distribution and MySQL server for you for the sake of brevity, but noted that Asterisk is in fact quite flexible in such matters. We now have a solid foundation on which to build our Asterisk system. In the following chapters we will explore how to connect devices to our Asterisk system in order to start placing calls internally, and work toward increasingly complex concepts in later chapters (such as video conferencing and WebRTC).

CHAPTER 4

Certificates for Endpoint Security

We only need to be lucky once. You need to be lucky every time.
—The IRA to Margaret Thatcher, after a failed assassination attempt

If you really want to do something, you'll find a way. If you don't, you'll find an excuse.
—Jim Rohn

The Inconvenience of Security

VoIP security can be regarded as two separate (but interconnected) challenges:

- Securing a system against toll fraud (which is generally the goal of SIP-based intrusion attempts)
- Securing a system against call interception (which relates to privacy, as well as improving toll fraud defenses)

There are of course many other aspects to the security of your system, but most of those are general security concepts, not specific to VoIP.

In this chapter we will focus on an area of security that is too often overlooked, namely the generation and application of certificates and keys in order to secure communication between endpoints and your system. In SIP communications, encryption is optional (and, unfortunately, not used most of the time). In WebRTC, it is required.

This chapter should by no means be considered the final word on securing your Asterisk system; there will be more covered in Chapter 22. We do hope, however, that it will provide you with a solid foundation on which to build a secure solution.

Securing SIP

If you build any sort of server that is exposed to the internet, and wait for a few short hours after powering it up, you will notice that the system will have already attracted probes attempting to determine if it has any vulnerable SIP services. You will then notice, a short while later, an increasing amount of attacks on the server attempting to compromise account security. Congratulations, your server has automatically been added to lists of SIP servers shared by criminals for the purpose of fraud. If one of these intrusions succeeds, the compromised platform will likely become part of an organized crime network, with you paying the bill for untraceable calls to various expensive destinations.

We're not playing around here; do not take your SIP security lightly or you are likely to find yourself the victim of a very expensive fraud attack.

Subscriber Names

The subscriber portion of the SIP credentials (the `userinfo` part of the URI) is far too often set to an extension number. This practice is fine for the purposes of addressing calls, but is not at all recommended for the purposes of authenticating an endpoint. The subscriber name of your endpoints should have no meaning outside of your organization. A MAC address is perfect for this. Without an actual address to probe, an attacker's job just got a lot more difficult.

You will see many SIP-type PBXs (including many based on Asterisk, unfortunately) that assign the extension number to the credentials of the phone. In this book, you will see that the extension number is not part of the phone credentials. There are several benefits to this, but from the perspective of security, the benefit is that if an attacker knows an extension, it does not provide any knowledge about how to authenticate a call through the system.

So, you might have a user with the SIP address `100@shifteight.org`, which your Asterisk system will associate with the device at `0000f3101010@yourpbx.com`. When a call is directed to `100`, it will ring `0000f3101010`, but the caller never knows anything about that endpoint.

You will see throughout this book that we will establish a relationship between an extension (which we believe is something that should be associated with a user or service) with a device identifier (which could be a SIP device or a phone number), and that a simple table can be used to tie them together (and subsequently increase both security and flexibility).

Secure SIP Signaling

By default, SIP messaging is passed in the clear, with no effective security. A man-in-the-middle attack is capable of obtaining all sorts of information about your calls. Transport layer security (TLS) is used to minimize this risk.

We'll talk more about how to configure devices to use TLS in the next chapter. All we need to do here is create the certificates.

There are three common ways to generate certificates. We'll provide examples for two of them (self-signed and LetsEncrypt), but will leave the exercise of obtaining formally issued certificates to the reader.

Self-signed certificates

The primary advantage of a self-signed certificate is that you don't have to validate it with any external entity. The disadvantage is that because of this, external entities will not trust it.

If you are securing SIP devices only for use within your controlled network environment, a self-signed certificate may be all you need. This is not considered the best approach, but in some cases it may be good enough, and it's generally better than nothing.

In this world full of automated crime, there is much to learn about privacy and security, and the cryptography necessary to both. However, this is a book about Asterisk, so what we are going to do is provide a template for generating the required components, and it'll be up to you to research further.

The `openssl` toolkit provides a tool that'll get the job done for us. We'll run it as follows:

```
$ sudo su asterisk -

$ mkdir /home/asterisk/certs

$ openssl req -x509 -nodes -newkey rsa:2048 -days 3650 \
-keyout /home/asterisk/certs/self-signed.key \
-out /home/asterisk/certs/self-signed.crt
```

You will be asked to provide some information, and then your key and cert will be written to the */home/asterisk/certs* folder.

You can add the following to the command to bypass the questions (change the information as appropriate to your situation):

```
-subj "/C=CA/ST=Ontario/L=Toronto/O=ShiftEight/CN=shifteight.org"
```

The full command should then look something like this:

```
$ openssl req -x509 -nodes -newkey rsa:2048 -days 3650 \
> -keyout /home/asterisk/certs/self-signed.key \
```

```
> -out /home/asterisk/certs/self-signed.crt \
> -subj "/C=CA/ST=Ontario/L=Toronto/O=ShiftEight/CN=shifteight.org"
```

This will generate a self-signed certificate and a private key, and save them both to */home/asterisk/certs/*. We can use them later when we are configuring our SIP endpoints.

It's probably a good idea to `chmod` your certs so only the relevant user/group can access them:

```
$ chmod 640 /home/asterisk/certs/*
```

Exit the `asterisk` user account.

```
$ exit

$ who am i # You should be astmin again.
```

There is an alternative to using self-signed certificates: if you have a domain name assigned to your server that can be reached from the public internet, you can generate a validated certificate using LetsEncrypt. Read on.

LetsEncrypt certificates

If you are interested in secure communications across the public internet (which you are, trust us), then having domain certificates provided by a certificate authority (CA) is useful.

The LetsEncrypt project provides free domain validation (DV) digital certificates. The free tool provided by the Let's Encrypt Foundation called `certbot` allows you to automate the obtaining and maintenance of trusted certificates.

At a minimum, your server will need a fully qualified domain name (FQDN) that maps to an external IP address that arrives at the machine. Any firewall in between will need to pass traffic for that hostname to the system you are obtaining the certificate for. If you cannot do this for whatever reason, the obtaining of a trusted certificate becomes more complex (and beyond the scope of this book).

`certbot` can be installed with `yum` as follows:

```
$ sudo yum -y install certbot
```

Once it's installed, you simply need to run the following:

```
$ sudo certbot certonly

How would you like to authenticate with the ACME CA?
-------------------------------------------------------------------------------
1: Spin up a temporary webserver (standalone)
2: Place files in webroot directory (webroot)
-------------------------------------------------------------------------------
Select the appropriate number [1-2] then [enter] (press 'c' to cancel): 1
```

If you've got a web server running, or are confident with option 2, that is OK, but these steps assume you don't have a web server running, and thus will need/want to use the built-in temporary web server that `certbot` will use to authenticate. This server is used to prove that you control the domain you're requesting a certificate for.

Answer the next questions as appropriate, and then at this point you will insert the hostname that you assigned to the IP address of your server:

```
Please enter in your domain name(s) (comma and/or space separated)  (Enter 'c'
to cancel): asteriskbook.shifteight.org
```

(replace *asteriskbook.shifteight.org* with the domain name you assigned).

`certbot` will perform its magic, and if all went well you should get some sort of message similar to the following:

```
IMPORTANT NOTES:
 - Congratulations! Your certificate and chain have been saved at:
   /etc/letsencrypt/live/asteriskbook.shifteight.org/fullchain.pem
   Your key file has been saved at:
   /etc/letsencrypt/live/asteriskbook.shifteight.org/privkey.pem
   Your cert will expire on 2018-07-23. To obtain a new or tweaked
   version of this certificate in the future, simply run certbot
   again. To non-interactively renew *all* of your certificates, run
   "certbot renew"
 - Your account credentials have been saved in your Certbot
   configuration directory at /etc/letsencrypt. You should make a
   secure backup of this folder now. This configuration directory will
   also contain certificates and private keys obtained by Certbot so
   making regular backups of this folder is ideal.
 - If you like Certbot, please consider supporting our work by:
   Donating to ISRG / Let's Encrypt:   https://letsencrypt.org/donate
   Donating to EFF:                    https://eff.org/donate-le
```

> Don't forget to donate to the Internet Security Research Group (ISRG) or the Electronic Frontier Foundation (EFF); they do important work and deserve our support.

You now have the certificates you'll need for enabling various TLS services on your system. We'll put them to use in the next chapter.

Not too difficult, eh?

Bear in mind that most certificates you get from an outside source will have an expiration date. In the case of LetsEncrypt, the current validity is three months. If you are going to put certificates into production, it is up to you to understand how to manage them (for example, automating renewal, which the LetsEncrypt folks have done a good job of simplifying).

Purchasing certificates from a formal certificate authority

If LetsEncrypt certificates do not provide the level of validation you require—for example, if you need organization validation (OV) or extended validation (EV)—you will need to obtain the services of a certificate authority that provides such things. These matters are beyond the scope of this book.

If you have worked through the examples for the self-signed and LetsEncrypt sections, you'll have at least a basic understanding of some of the process of obtaining certificates from a certificate authority, as many of the steps will be similar.

Securing Media

The certificates we've obtained can be used to secure both our signaling, and the payload itself (i.e., what's being spoken, or the video being transmitted). Note that the mechanisms to secure signaling are an SIP protocol thing, and the mechanisms to secure the media are an RTP protocol thing. Keep in mind that encrypting your SIP signaling does not mean you're automatically also encrypting your media (RTP) traffic.

Encrypted RTP

Encrypting the Real Time Protocol will achieve the effect of securing our media streams.

There are two mechanisms commonly used to provide media encryption: SDES and DTLS-SRTP. SDES is a media encryption mechanism that trusts that the signaling is secure. In other words, if you are using TLS to secure your SIP signaling, then SDES is likely how your media encryption is being handled.

DTLS-SRTP, on the other hand, does not trust the signaling. It is important because the WebRTC standard requires that media be encrypted this way.

The certificates we've generated here should work in both scenarios. In upcoming chapters, when we are configuring SIP or WebRTC endpoints, we will cover in more detail how to use the certificates. For now, it's enough that we've generated the certificates and have them available for use.

Conclusion

Make no mistake: security makes everything more complicated. In the good old days, you could fire up a SIP connection with a half-dozen lines of config and call it a day. That doesn't fly anymore, and while that type of configuration will still work (simply use UDP instead of TLS, and all you need is a password), we decided that starting with this edition, all configuration examples will choose more secure options

wherever possible. We're not claiming to present the final word on VoIP security, but we are going to deliver examples that pay more than lip service to the concept.

Next up, we'll discuss how to configure endpoints on your Asterisk system (using the keys and certificates we've just generated).

CHAPTER 5
User Device Configuration

I don't always know what I'm talking about, but I know I'm right.
—Muhammad Ali

It's time to connect some SIP user devices to Asterisk. While we are going to focus on the Asterisk end of things, keep in mind that defining a channel in Asterisk for the device to connect through is only half of the configuration; you also need to configure the other end—the device itself (usually a phone)—so it knows where to send its calls.

An Important Note About SIP Endpoints

Configuring the other end of the SIP relationship is of course necessary, but is not part of Asterisk configuration, and ultimately is out of the scope of this book. There must be thousands of different types of SIP endpoints in the market, including desk phones, softphones, PBXs, proxy servers, conferencing servers, and all manner of other products. Each manufacturer has its own tools to allow you to configure its products (and some of them require extensive knowledge). SIP is a complex protocol. Having said that, most SIP desk telephones have some sort of web interface, and most softphones have a configuration menu built into their GUI.

At its simplest, configuring a SIP device involves providing three parameters:

- Address of the server it's going to connect through (your Asterisk server)
- Username (which could also be called the subscriber name, extension, or something similar)
- Password

While each type of endpoint is going to be different, they'll all follow a similar convention, and while there are potentially hundreds of configuration options, it's quite common to only configure those three things.

In other words, there are two separate tasks needed to configure a device to work with Asterisk:

- Telling Asterisk about the device (configuring the channel credentials in Asterisk)
- Telling the device about Asterisk (accessing the configuration tools for the device, and telling it where its server is and how to connect)

Some Thoughts About SIP

SIP is a peer-to-peer protocol, and while it is common to have a setup where endpoints (telephones) behave as clients, and some sort of gateway (such as Asterisk) provides routing and features, the protocol itself works in terms of peer-to-peer relationships (Figure 5-1). Two SIP endpoints can talk directly to each other (i.e., a pair of SIP telephones should in theory be able to create a sort of "intercom" directly between each other without a PBX in the middle).

That being said, by far most SIP transactions happen through a server of some sort, which typically remains in the call path and bridges all connections (also not required by the protocol). When a SIP call is made from a telephone to another telephone through Asterisk, there are actually two calls happening: a call from the originating set to Asterisk, and another separate call from Asterisk to the destination set (this second leg of the call might not even use SIP). Asterisk bridges the two together internally. Similarly, if you are making an "external" call, Asterisk will accept the call from your set, and then send a call out another channel that would be considered a trunk, and again would bridge those channels together. At the protocol level, a set-to-set and set-to-trunk call look very similar.

Using a SIP phone with Asterisk means that you will want to configure the SIP telephone to send all its calls to Asterisk, even though the device is quite capable of directly connecting to another SIP endpoint without the Asterisk server. The phone will treat Asterisk as its registrar and proxy server (even though Asterisk is in fact a Back to Back User Agent, or B2BUA) and will look to Asterisk for routing decisions for all calls.

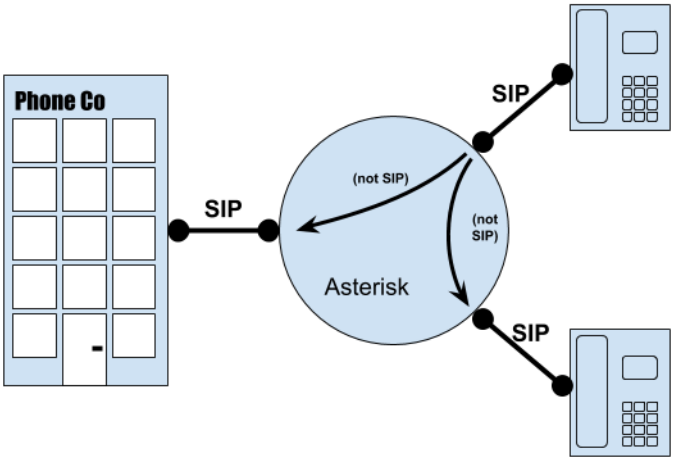

Figure 5-1. Asterisk is a gateway

Some Thoughts About Set Provisioning

While most devices will have a web-based interface for defining parameters, if you're putting more than one or two phones into production you should look into using a server-based provisioning process of some sort. In this way, sets will connect to the server, identify and authenticate, and download customized files that contain their parameters (it is very common to use the MAC address of the telephone as an identifier for naming each unique config file). Configuration files for various products are commonly served up by an HTTPS or SFTP server, and will be formatted as XML or some form of key/value pairs.

Unfortunately, the exact download process, protocol, and syntax of these files will differ from manufacturer to manufacturer. It's not difficult to learn if you're familiar with such concepts, but to attempt to cover all of them (and keep ever-changing processes up to date) would be impossible. Manufacturers usually offer freely downloadable and detailed configuration guides for their telephones, so with a bit of research and familiarity with configuring file services on Linux, you will find a wealth of information is available online. In our experience the documentation provided by the manufacturers is generally excellent. It will represent the most up-to-date information on provisioning their devices.

When you are troubleshooting set provisioning, always test the download using a computer first. If you can't download the files on your computer, your sets probably won't be able to download them either.

We will say one final word about this: make sure whatever process you use includes encryption of the config files, so that if the files are stolen, only the intended recipient is capable of decrypting them. Most manufacturers have done a good job of making this a fairly simple thing to achieve. Don't send unencrypted configuration files across the public internet.

In this chapter we focus on the configuration of sets from the perspective of Asterisk, so we're not going to say too much about bootstrapping phones; you are going to need to do your own research in that regard. We'll be using a couple of softphones in our lab, and you can too. Our examples will attempt to provide enough information that you'll be able to configure whatever SIP devices you are using. If we can get you through registering a couple of softphones on your lab system, we'll have set you on the path toward more complex scenarios (which will typically require some research and prototyping on your part).

Telephone Naming Concepts

Before we get started with configuring Asterisk for our telephones, we are going to recommend some best practices regarding telephone naming.

First up, you should not assign your telephones an extension number; instead, design the system so that the extension number is assigned to the user, and then assign telephones or other resources to that user. The telephones themselves should be named in reference to something unique to them, such as a MAC address or computer name. In a flexible, next-generation PBX, you want to abstract the concepts of users, extension numbers, and telephones, so as to have the most flexibility and ease of management.

In Asterisk, there is really no concept of a user at all. Extensions are triggers that initiate a sequence of instructions. Yes, you might write a bit of dialplan specifying that when extension number 100 is dialed, Asterisk will ring the phone on your desk. However, extension 100 could just as easily access a company voicemail box, or perhaps play back a prompt, join a conference room, or do any number of other things. We can even write dialplan that specifies that extension 100 should ring the device on your desk from Monday to Friday between 9 A.M. and 5 P.M., but ring a device on someone else's desk the rest of the time. Inversely, when a call is made from a device during business hours, the caller ID could show a daytime number, and the rest of the time could show an after-hours number (many reception desks become security desks at night).

In Asterisk, extensions are not phones. Therefore, don't give your telephones identifiers that are extensions.

Asterisk Extensions

The concept of an extension in Asterisk is crucial. In most PBXs, an extension is a number that you dial to cause a phone or service to ring. In Asterisk, an extension is the name of a grouping of instructions in the dialplan. Think of an Asterisk extension as a script name, and you're on the right track. Yes, an Asterisk extension could be a number that rings a phone, but it could just as easily be a word (such as `voicemail`) that provides a simple service of some sort, without ever sending the call out any channel.

We'll be going into Asterisk extensions in far more detail throughout this book, but before we do that we want to get some phones set up.

The abstraction between the name of an extension and the telephone that extension might ring is a powerful concept. An excellent example of this is a PBX feature commonly known as *hot-desking*, which allows users to share a desk and/or move around to different desks. Let's say we have three sales agents who typically work outside of the office, but spend a couple of days each month in the office to do paperwork. Since they are unlikely to be on-site at the same time, instead of each agent having a separate telephone, they could share a single office phone (or on a larger scale, a dozen folks could share a pool of, say, three phones). This scenario illustrates the convenience (and necessity) of allowing the system to separate the concept of a user and extension from the physical phone.

The best way to name a physical desk phone (or any physical SIP endpoint) is using the MAC address of the device, which is unique to the phone, follows it where it goes, and doesn't require configuration changes to the phone if the user changes. Some corporations have stickers they place on their equipment with a bar code and other information that allows them to keep stock of provisioned equipment; these unique codes would also be an acceptable choice to use for phone names, as they don't provide any logical relation to a particular person, but do provide specific information about the devices themselves.

Softphones on laptops can also use a MAC address or serial number, but make sure the set name includes a reference to the fact that it's a softphone. `[JIM_VANM_SOFT]` is a decent name, but `[JIMS_PHONE]` is not. If the softphone is running on a desk computer (i.e., it's not going to move around with a user), then name it using the convention you use for your computers (`[DESK-5F23-SOFT]` or `[CUST_SRVC_001_SOFT]` are potentially good names).

The choice is yours as to how you want to name your phones. Our goal is simply to help you understand why the best practice is to abstract any concept of the telephone being owned by a person. A phone is just a way to get audio to and from a human, and signal back and forth, so it's far better to make it possible to mix and match them as users move around, and people come and go.

Throughout this book, you'll see us using phone names that look like MAC addresses (such as `[0000F3000001]` and `[0000F3000002]`), or generic desk names (`[DESK-001-SOFT]`, `[DESK-002-SOFT]`) to differentiate between devices. You will generally want to use phone names that match the hardware you are using (or some other string that is unique to the device you are registering).

As a final consideration, we should make it clear that what we are suggesting regarding device names is not a technical requirement. You are free to name your devices anything you want, as long as they meet the requirements of Asterisk's naming conventions for devices (stay with alphanumeric characters with no spaces and you'll be fine).

You'll see plenty of Asterisk systems that tie the device name to the extension of the user, but we're not fans of this method.

Hardphones, Softphones, and ATAs

There are three types of endpoints you would typically provide your users to serve as a telephone set. They are popularly referred to as hardphones (or desk phones), softphones, and Analog Terminal Adaptors (ATAs).

A *hardphone* is a physical device—an office telephone. It has a handset, numbered buttons, a screen of some sort, and so on. It connects directly to the network and is probably what people are referring to when they talk about a VoIP telephone (or a SIP telephone). It's normally going to sit on your desk, but it could be mounted on a wall, in an elevator, at a side-table, or in a box by the side of the road.

A *softphone* is a software application that runs on a laptop, desktop, smartphone, or other computing device. The audio must pass through the device's sound system, so you normally need a headset that will work well with telephony applications. Softphone applications have become popular with smartphones because they allow you to connect to telephone networks other than just the cellular network (for example, you can register as an extension on your corporate PBX). The interface of the softphone is often styled to look like a physical telephone, but this is not necessary. WebRTC will allow all sorts of additional capabilities in this area, as it essentially allows a softphone to simply be part of a session within a browser. To the PBX, the softphone looks and behaves exactly the same as a hardphone.

An *ATA* is designed to allow traditional analog telephones (and other analog devices, such as fax machines, cordless phones, paging amplifiers, and such) to connect to a SIP network,[1] and will typically be a sandwich-sized box that contains an RJ-11 connector for the phone (commonly referred to as an FXS port), an RJ-45 connector for the network, and a power connector. Some ATAs may support more than one phone. Other ATAs may have advanced features in them such as a firewall or an FXO port (an analog port that can connect to a PSTN circuit). From the perspective of the PBX, the ATA looks exactly like a SIP telephone.

Hardphones have the advantage that the handsets have good acoustic properties for voice communications. Any decent-quality telephone is engineered to pick up the frequencies of the human voice, filter out unwanted background noise, and normalize the resulting waveform. People have been using telephones for as long as the telephone network has existed, and we tend to like what is familiar, so having a device that communicates with Asterisk using a familiar interface will be attractive to many users. Also, a hardphone does not require your computer to be running all the time.

Disadvantages to hardphones include the fact that they are not easily portable, and they are expensive relative to the many quality softphones on the market today that are available for free. Also, the extra clutter on your desk may not be desirable if you have limited work space. If you move around a lot and are not generally at the same location, a hardphone is not likely to suit your needs (although, one at each location you regularly visit might be a valid solution).

Softphones solve the portability issue by being installed on a device that is likely already moving with you, such as your laptop or smartphone. Also, their minimal cost (typically free, or around $30 for a fully featured one) is attractive. Since many softphones are free, it is likely that the first telephone set you connect to Asterisk will be a softphone. Also, because softphones are just software, they are easy to install and upgrade, and they commonly have other features that utilize other peripherals, like a webcam for video calling, or perhaps an ability to load files from your desktop for faxing. Another potentially huge advantage of a softphone is that it is often possible to integrate them with other applications running on the device.

Some of the disadvantages of softphones are the not-always-on nature of the devices, the necessity to put on a headset each time you take a call, and the fact that many PCs will at random times during the day choose to do something other than what the user wants them to do, which might cause the softphone to stop working while some background task hogs the CPU. In a mobile device, the softphone can consume resources, affecting battery life, performance, and operating expense.

1 Or any other network, for that matter. ATAs could more formally be said to be analog-to-digital gateways, where the nature of the digital protocol may vary (e.g., proprietary ATAs on traditional PBXs). Point being, an ATA is not necessarily a SIP device.

ATAs have the advantage of allowing you to connect analog devices to your SIP network,[2] such as cordless phones (which are still superior in many cases to more advanced types of wireless phones, and far less expensive[3]), paging amplifiers, ringers, and antique telephones.[4] ATAs can also sometimes be used to connect to old wiring, where a network connection might not function correctly, or to outbuildings (such as a gatehouse), where a standard ethernet connection would never reach.

The main disadvantage of an ATA is that you will not get the same features through an analog line as you would from a SIP telephone. This is technology that is over a century old.

With Asterisk, we don't necessarily need to make the choice between having a softphone, a hardphone, or an ATA; it's entirely possible and quite common to have a single extension number that rings multiple devices at the same time, such as a desk phone, the softphone on a laptop, a cell phone, and perhaps a strobe light in the back of the factory (where there is too much noise for a ringer to be heard).

More than any other endpoint, the softphone is set to evolve into something far more encompassing than a simple telephone application. The emergence of WebRTC may finally deliver that which has been predicted for many long years: the integration of real-time voice into computing (specifically web-based) applications. There are of course many ways to achieve this already, but WebRTC's advantage is that it is an open standard, built right into all browsers with no plug-ins required. The softphone is dead; long live the softphone.

We still like a desk phone for regular telephone calls, though.

Configuring Asterisk

In this section we'll cover how to configure PJSIP to handle various SIP endpoints. This was traditionally done by editing files in the */etc/asterisk/* directory; however, we have elected to demonstrate how this is done through a database, as it is a generally superior method, especially as a system grows. If you are more comfortable with

2 An ATA is not the only way to connect analog phones. Hardware vendors such as Digium and Sangoma sell cards that go in the Asterisk server and provide analog telephony ports. Larger installations can also use channel banks or MSANs; however, this method of connecting legacy telecom circuits is a more advanced subject, and not the focus of this book.

3 For a really awesome cordless analog phone, you want to check out the EnGenius DuraFon devices, which are expensive, but impressive.

4 Our friend Brian Capouch has mashed together many entertaining demonstrations of how antique telephone hardware can be made to work with Asterisk.

using *.conf* files, you should find that fairly easy to do once you have the basics figured out.[5]

Asterisk allows devices using many different protocols to speak to it (and therefore to each other), but `chan_pjsip` is the only VoIP module that is still actively maintained; the rest are legacy code. You aren't likely to have any use for other VoIP protocols (such as IAX2, Skinny/SCCP, Unistim, H.323, and MGCP). Those protocols have an historical significance, since it was in large part due to the fact that Asterisk would talk to anything and everything that it had such an impact on the telecom industry. Nowadays, however, SIP has pretty nearly replaced everything, so those channel drivers are now historical curiosities, and nothing more. If you are still interested in one of those other protocols, focus on getting comfortable working with SIP first, and recognize that you're going to be pretty much on your own.

The channel configuration tables in the database[6] contain the configuration details relevant to that channel driver, as well as the parameters and credentials specific to the SIP devices and providers you wish to connect to Asterisk (incoming and outgoing). To put that more simply: all calls in and out of Asterisk must pass through a channel.

Most parameters have defaults, which you will find documented in the sample files. Start with reading the *pjsip.conf.sample* file found in your *~/src/asterisk.<TAB>/configs/samples/* folder. It will provide plenty of information about defaults, as well as information about other resources worth reading. We will not use the file for the actual configuration (we are using the database instead); however, the file is an excellent reference, and you should keep it near at hand, as it will have the answers to many questions you may have about parameters.

We are going to focus on getting a basic device going for you as simply as possible. We have found that setting up channels is one of the more frustrating things new Asterisk users experience, and we want to demonstrate that at its most basic level, it does not need to be painful at all. Once you have succeeded here, you'll always have a known-good configuration to fall back on, as you move forward into more complex scenarios.

5 Put simply, you would need to find the *pjsip.conf.sample* file, and use it as a template to create a *pjsip.conf* file in your */etc/asterisk* folder, and then edit that file in a manner similar to how we're going to do things in the database.

6 Or in the *.conf* file, if you choose to go that route.

How Channel Configuration Works with the Dialplan

Channels are how Asterisk connects calls to everything outside of it, but it is the dialplan that defines what happens to calls as they pass through the system. Therefore, channels and dialplan are inextricably linked. The dialplan is the heart of an Asterisk system: it controls how call logic is applied to any connection, from any channel, such as what happens when a device dials extension 101, or an incoming call from an external provider is routed to a DID. The PJSIP channel configuration tables in the database, plus the */etc/asterisk/extensions.conf* file, will play a critical role in most—if not all—calls routed through the system. Once you have your channels configured, you will find that most of your work will be in the dialplan. We will dive deeply into this in upcoming chapters.

When a call comes into Asterisk, the identity of the incoming call is matched in the channel configuration for the protocol in use (SIP connections are handled by the PJSIP channel driver). The channel driver will handle authenticating the incoming connection. The channel configuration also *defines where that channel will enter the dialplan*.

Once Asterisk has determined how it will handle the channel (i.e., it's authenticated and the various parameters for the call have been established), it can pass call control to the correct context in the dialplan. It is the `context` parameter in the `ps_endpoints` table that tells the channel where the call will enter the dialplan (which contains all the information about how to handle and route the call).

In Figure 5-2, we see that the call flow through the configuration, for an internal call (set-to-set), will look something like this:

1. User of set `0000f30A0A0101` dials `102`.
2. Asterisk matches the incoming SIP request against an endpoint (and authenticates it).
3. The number dialed is matched to the `[sets]` context in the dialplan.
4. The `Dial()` application is used to send a call out a PJSIP channel, to the contact associated with `0000f30B0B02`.
5. The contact address is determined (typically based on registration if it's a set, but could be hardcoded as well if it is a trunk).
6. A SIP INVITE message is sent to the destination.

A key point to remember is that the channel configuration files control not only how calls enter the system, but also how they leave the system. So, for example, if one set calls another set, the channel configuration file is used not only to pass the call into the dialplan, but also to direct the call out from the dialplan to the destination.

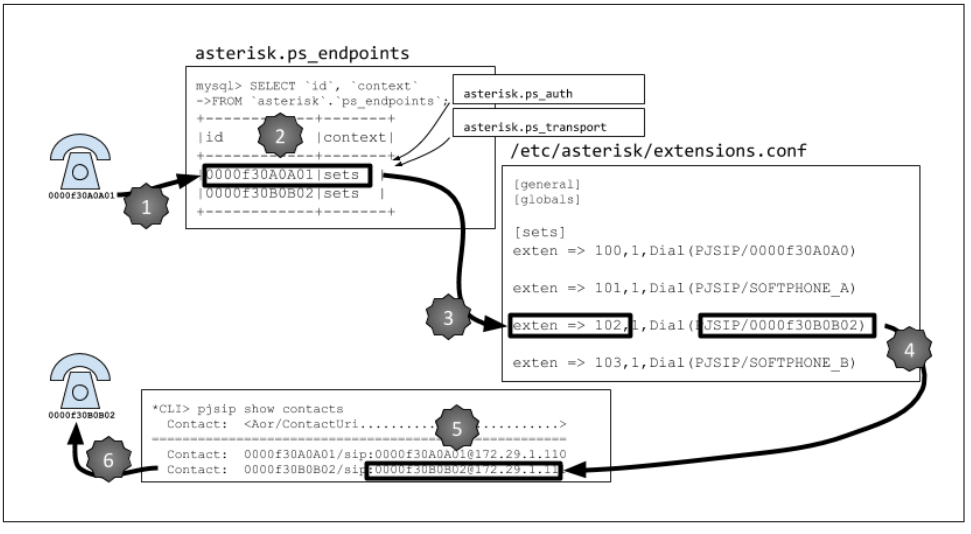

Figure 5-2. Relationship of pjsip.conf to extensions.conf

chan_pjsip

The PJSIP channel module is one of the newer modules in Asterisk. It replaced the original `chan_sip` module.

The old SIP module, `chan_sip`, has been deprecated, so we will not be documenting it in this book. If you are new to Asterisk, you should stick to PJSIP, but it may be helpful to understand that `chan_sip` was around for many years, and is still widely used on old systems.

The PJSIP framework, as implemented in Asterisk, is composed of many components. If you check your database, you will find that there are over a dozen tables relating to PJSIP (prefixed with `ps_`). Not all of them relate to set configuration, though.

PJSIP is an open source, all-purpose multimedia communication library that provides not only SIP signaling, but also the media features and NAT traversal functions that are essential components of a SIP-based application. It is provided and supported by Teluu Ltd., and the library is used in far more than just Asterisk. Softphones, proprietary products, and other open source projects also make use of the framework. The Asterisk community needed/wanted a new SIP channel driver, and rather than building one from scratch, the developers decided to use the PJSIP library.

The components listed in Table 5-1 will be used in your endpoint configurations.

Table 5-1. PJSIP components of Asterisk

Component	Purpose
`ps_aors`	The Address Of Record (AOR) table is used to define how Asterisk can contact an endpoint. When the set attempts to register, Asterisk will consult the AOR in order to identify it.
`ps_auths`	The Authentication section contains the credentials that endpoints will need to provide in order to authorize communication with Asterisk.
`ps_contacts`	Typically created automatically as part of the registration process, it is here that Asterisk will store the details of the endpoint determined during registration.
`ps_endpoints`	The heart of the SIP configuration, it is here that each endpoint is defined. This is also where the associations to the other PJSIP records are defined.

Adding an endpoint

During the installation, several example endpoints were created for you, in order to simplify the process of providing you with a working system. If you want to add additional endpoints, you simply need to define additional records in each of the `ps_aors`, `ps_auths`, and `ps_endpoints` tables.

Let's say we want to add a couple of softphones to our system, named `SOFTPHONE_A` and `SOFTPHONE_B`.

First, into the `ps_endpoints` table, we'll want to add the following:

```
$ mysql -D asterisk -u asterisk -p
```

Let's review what we have there already (from the previous chapters):

```
mysql> select id,transport,aors,auth,context,disallow,allow from asterisk.ps_endpoints;
+-------------+--------------+-------------+-------------+----------+----------+-------+
| id          | transport    | aors        | auth        | context  | disallow | allow |
+-------------+--------------+-------------+-------------+----------+----------+-------+
| 0000f30A0A01 | transport-udp | 0000f30A0A01 | 0000f30A0A01 | starfish | all      | ulaw  |
| 0000f30B0B02 | transport-udp | 0000f30B0B02 | 0000f30B0B02 | starfish | all      | ulaw  |
+-------------+--------------+-------------+-------------+----------+----------+-------+
2 rows in set (0.00 sec)
```

We're going to insert a couple of extra records.

```
mysql> insert into asterisk.ps_endpoints
(id,transport,aors,auth,context,disallow,allow)
values
('SOFTPHONE_A','transport-tls','SOFTPHONE_A','SOFTPHONE_A','sets','all','ulaw'),
('SOFTPHONE_B','transport-tls','SOFTPHONE_B','SOFTPHONE_B','sets','all','ulaw');
Query OK, 2 rows affected (0.02 sec)
```

The `ps_endpoints` table should then look something like this:

```
mysql> select id,transport,aors,auth,context,disallow,allow from ps_endpoints;
+--------------+--------------+--------------+--------------+---------+----------+-------+
| id           | transport    | aors         | auth         | context | disallow | allow |
+--------------+--------------+--------------+--------------+---------+----------+-------+
| 0000f30A0A01 | transport-udp | 0000f30A0A01 | 0000f30A0A01 | sets    | all      | ulaw  |
| 0000f30B0B02 | transport-udp | 0000f30B0B02 | 0000f30B0B02 | sets    | all      | ulaw  |
| SOFTPHONE_A  | transport-tls | SOFTPHONE_A  | SOFTPHONE_A  | sets    | all      | ulaw  |
| SOFTPHONE_B  | transport-tls | SOFTPHONE_B  | SOFTPHONE_B  | sets    | all      | ulaw  |
+--------------+--------------+--------------+--------------+---------+----------+-------+
4 rows in set (0.00 sec)
```

Then, we'll need two related records in the `ps_aors` table:

```
mysql> insert into asterisk.ps_aors (id,max_contacts)
values ('SOFTPHONE_A',2),('SOFTPHONE_B',2);
Query OK, 2 rows affected (0.01 sec)
```

The `ps_aors` table should then return the following:

```
mysql> select id from asterisk.ps_aors;
+--------------+
| id           |
+--------------+
| 0000f30A0A01 |
| 0000f30B0B02 |
| SOFTPHONE_A  |
| SOFTPHONE_B  |
+--------------+
4 rows in set (0.00 sec)
```

Finally, the `ps_auths` table will need records for each new device.

```
insert into asterisk.ps_auths (id,auth_type,password,username)
values ('SOFTPHONE_A','userpass','iwouldnotifiwereyou','SOFTPHONE_A'),
('SOFTPHONE_B','userpass','areyoueventrying','SOFTPHONE_B');

Query OK, 2 rows affected (0.00 sec)
```

And, if all went well, you will have the following authorization records:[7]

```
mysql> select id,auth_type,password,username
    -> from asterisk.ps_auths;
+--------------+-----------+---------------------+--------------+
| id           | auth_type | password            | username     |
+--------------+-----------+---------------------+--------------+
| 0000f30A0A01 | userpass  | not very secure     | 0000f30A0A01 |
| 0000f30B0B02 | userpass  | hardly to be trusted | 0000f30B0B02 |
| SOFTPHONE_A  | userpass  | iwouldnotifiwereyou | SOFTPHONE_A  |
| SOFTPHONE_B  | userpass  | areyoueventrying    | SOFTPHONE_B  |
+--------------+-----------+---------------------+--------------+
4 rows in set (0.00 sec)
```

The new endpoints are now ready to have devices register to them. You can verify that they exist with the following Asterisk CLI command:

7 Except, of course, that your passwords will be far better than the ones we've used here.

```
mysql> exit
$ sudo asterisk -rvvvvv
*CLI> pjsip show endpoints
```

The output should list your new endpoints:

```
 Endpoint:  0000f30A0A01                                          Unavailable   0 of inf
     InAuth:  0000f30A0A01/0000f30A0A01
        Aor:  0000f30A0A01                                        2
  Transport:  transport-udp             tls      0      0  0.0.0.0:5061

 Endpoint:  0000f30B0B02                                          Not in use    0 of inf
     InAuth:  0000f30B0B02/0000f30B0B02
        Aor:  0000f30B0B02                                        2
     Contact:  0000f30B0B02/sip:0000f30B0B02@172.29.1.110 7542ca7ce1 Unknown         nan
  Transport:  transport-udp             udp      0      0  0.0.0.0:5060

 Endpoint:  SOFTPHONE_A                                           Unavailable   0 of inf
     InAuth:  SOFTPHONE_A/SOFTPHONE_A
        Aor:  SOFTPHONE_A                                         2

 Endpoint:  SOFTPHONE_B                                           Unavailable   0 of inf
     InAuth:  SOFTPHONE_B/SOFTPHONE_B
        Aor:  SOFTPHONE_B                                         2
```

Hang on a minute…

Do you notice that there's no `transport` defined for your new endpoints? That's because we haven't defined anything yet for TLS; we've just configured for bog-standard UDP-style SIP.

Since we have those fancy keys we generated in the previous chapter, let's implement them now and see if we can fix this.

```
$ sudo vim /etc/asterisk/pjsip.conf

[transport-udp]
type=transport
protocol=udp
bind=0.0.0.0

[transport-tls]
type=transport
protocol=tls
bind=0.0.0.0
cert_file=/home/asterisk/certs/self-signed.crt       #if you used certbot, the location
priv_key_file=/home/asterisk/certs/self-signed.key   #of those keys goes here
```

Now, because we're putting some files in a folder that wasn't part of our SELinux config, we have to fix that. What we want is to get SELinux to generate an error, so we're going to reload the `res_pjsip.so` module, even though it will fail to load the `transport` correctly:

```
*CLI> module reload res_pjsip.so
```

Now we should have the errors we want, so we're going to search for them in the system log.

```
$ sudo grep -i sealert /var/log/messages |egrep "cert|crt"
```

You'll see some messages that look similar to this (we've trimmed them for brevity):

```
SELinux is preventing ... on the file /home/asterisk/certs/self-signed.crt.
 For complete SELinux messages run: sealert -l 1dbe51e2-7321-41d3-a5bb-f8f1b4a6f787

SELinux is preventing ... on the directory certs.
 For complete SELinux messages run: sealert -l 879db542-e0a9-43e8-8763-62fcf068bfee

SELinux is preventing ... on the file self-signed.crt.
 For complete SELinux messages run: sealert -l 8fb85940-ee82-44bd-adcb-e30d31ee516a
```

What's useful is that SELinux is telling you exactly what you need to do to solve the problem!

For each of the three messages relating to access to the certs we just configured, run the associated command. We'll just do one to show you what we mean, but you may need to run more than one until it loads clean.

```
$ sealert -l 8fb85940-ee82-44bd-adcb-e30d31ee516a
```

You're going to get something like this:

```
SELinux is preventing asterisk from read access on the file self-signed.crt.
*****  Plugin catchall (100. confidence) suggests   **************************
You can generate a local policy module to allow this access.
allow this access for now by executing:
# ausearch -c 'asterisk' --raw | audit2allow -M my-asterisk
# semodule -i my-asterisk.pp
```

You're not `root`, but you'll run both the commands it specifies:

```
$ sudo ausearch -c 'asterisk' --raw | audit2allow -M my-asterisk

$ sudo semodule -i my-asterisk.pp
```

Remember how everybody is telling you to just disable SELinux? Well, you don't have to do that anymore.

OK, restart Asterisk (`$ sudo service asterisk restart`) and make sure your log-file isn't generating SELinux errors (ignore the *.odbc.ini* errors as they don't relate to */etc/odbc.ini*, and shouldn't affect anything).

You should see that `transport-tls` is now ready to use:

```
*CLI> pjsip show transports
Transport:  <TransportId........>  <Type>  <cos>  <tos>  <BindAddress.....................>
===============================================================================================
Transport:  transport-tls             tls      0      0  0.0.0.0:5061
Transport:  transport-udp             udp      0      0  0.0.0.0:5060
```

If you find it's still not loading, go back and work the SELinux errors in the */var/log/messages* file. Sometimes more than one has to be dealt with.

Testing to Ensure Your Devices Have Registered

Once your devices have registered to Asterisk, you will be able to query the location and state of them from the Asterisk CLI.

It is a common misconception that registration is how a device authenticates itself for the purpose of obtaining permission to make calls. This is incorrect. The only purpose of registration is to allow a device to identify its location on the network, so that Asterisk[8] knows where to send calls intended for that device.

Authentication for outgoing calls is an entirely separate process and always happens on a per-call basis, regardless of whether a set has registered. This means that your set may be able to make calls, but not receive them. The most common cause of this is a firewall that has closed the incoming port (and the solution is to set a registration timer that's low enough that it will re-register every few minutes, so that the firewall will keep the relevant SIP port open).

It is possible to have a set register successfully, and yet still not be allowed to make calls. Point being, just because it's registered doesn't mean it can make calls (although this will almost always be the case).

Verifying the registration of a set is the simplest way to verify that you have configured it correctly.

Remember that configuration of the set does *not* happen in Asterisk. You have to configure the device using whatever tools the manufacturer has provided.

To check the registration status of a device, simply call up the Asterisk CLI:

```
$ sudo asterisk -rvvvv
```

Typing the following command returns a listing of all the peers that Asterisk knows about (devices that have registered will have a corresponding `Contact`):

```
*CLI> pjsip show aors
      Aor:  <Aor.........................................>  <MaxContact>
    Contact:  <Aor/ContactUri.........................> <Hash....> <Status> <RTT(ms)..>
==========================================================================================
      Aor:  0000f30A0A01                                  2
```

8 Or any other SIP registrar server, for that matter.

```
        Aor:  0000f30B0B02                                        2
    Contact:  0000f30B0B02/sip:0000f30B0B02@172.29.1.110:5 7542ca7ce1 Unknown         nan

        Aor:  SOFTPHONE_A                                         2

        Aor:  SOFTPHONE_B                                         2
```

A Basic Dialplan to Test Your Devices

In the next chapter we're going to dive into the Asterisk dialplan. Here, we're going to lay down a very simple dialplan so that if you register devices to the various SIP endpoints already in the PJSIP configuration, you should be able to make test calls between them.

```
$ sudo -u asterisk vim /etc/asterisk/extensions.conf

[general]
[globals]

[sets]
exten => 100,1,Dial(PJSIP/0000f30A0A01)

exten => 101,1,Dial(PJSIP/0000f30B0B02)

exten => 102,1,Dial(PJSIP/SOFTPHONE_A)

exten => 103,1,Dial(PJSIP/SOFTPHONE_B)

exten => 200,1,Answer()
     same => n,Playback(hello-world)
     same => n,Hangup()
```

From your Asterisk console type the following command:

```
*CLI> dialplan reload

*CLI> dialplan show
```

You will see that in the context `sets` there are some extension numbers you can call.

This basic dialplan will allow you to dial your SIP devices using extensions 100, 101, 102, and 103. You can also listen to the `hello-world` prompt that was created for this book by dialing extension 200.

Register a couple of SIP phones (you can download a softphone to your PC and another one to a tablet or smartphone). You should be able to dial between your extensions. Open up the CLI in order to see the call progression. You should see something like this (and the set you are calling should ring):

```
-- Executing [102@sets:1] Dial("PJSIP/SOFTPHONE_A-00000001", "PJSIP/0000f30B0B02")
-- Called PJSIP/0000f30B0B02
-- PJSIP/0000f30B0B02-00000002 is ringing
```

If this does not happen, review your configuration and registration and ensure you have not made any typos.

Registering a Device to Asterisk

There are so many different sorts of SIP devices you can register to Asterisk, it's impossible to provide an example that'll be useful to everyone. You could have a PC, or a Mac, or a Linux workstation, or an iPhone, or an Android, or a SIP desk phone, or some other SIP device entirely; each type of device has many different sorts of SIP clients available, and they're all mostly the same, but just different enough to be annoying to a novice.

We can't tell you the specific method to register, but we have found that of the dozens, or perhaps hundreds, of options in each device, the basic process is similar for all of them. You will need to provide:

- The address of your Asterisk server (`hostname`, `domain`, `proxy`, and `server` are all fields we've seen for this)
- The identity of the device (`id`, `user`, `subscriber`, `username`, `extension`, `name`, etc.)
- A password

If it's getting complicated, you've probably gone too deep. Keep it simple. If the product doesn't have good documentation, then it might not be the right product for you. For most SIP phones you can find instructions on the web for how to register them to Asterisk.

If you register a second device (you have four of them now!), you can make test calls between them.

Spend a bit of time on this and make sure you understand it all. It's all critical to everything that follows.

Under the Hood: Your First Call

In order to get you thinking about what is happening under the hood, we're going to briefly cover some of what is actually happening with the SIP protocol when two sets on the same Asterisk system call each other.

Asterisk as a B2BUA

Bear in mind that there are actually two calls going on here: one from the originating set to Asterisk, and another from Asterisk to the destination set. SIP is a peer-to-peer protocol, and from the perspective of the protocol there are two calls happening. The SIP protocol is not aware that Asterisk is bridging the calls; each set understands its connection to Asterisk, with no real knowledge of the set on the other side. It is for this reason that Asterisk is often referred to as a B2BUA (Back to Back User Agent). This is also why it is so easy to bridge different protocols together using Asterisk.

For the call you just made, the dialogs shown in Figure 5-3 will have taken place.

For more details on how SIP messaging works, please refer to the SIP RFC.

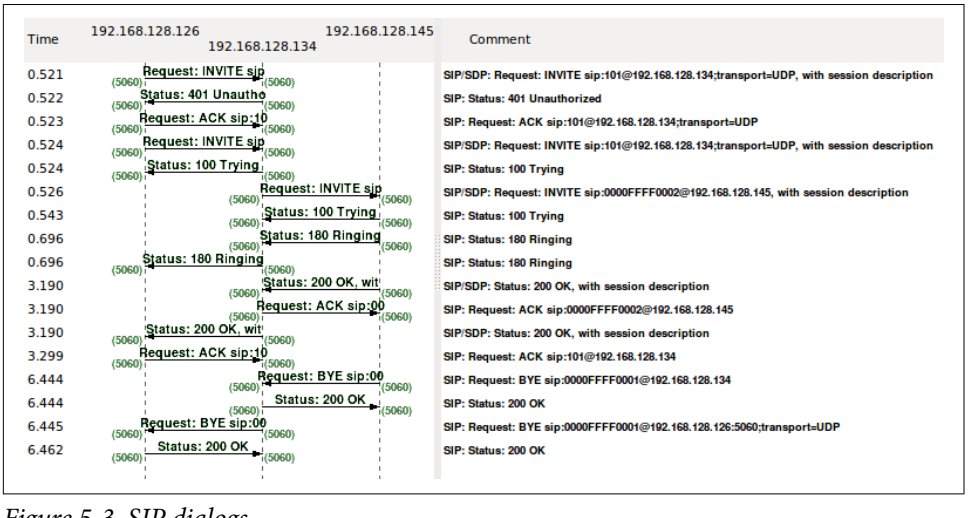

Figure 5-3. SIP dialogs

Conclusion

In this chapter you learned best practices for device naming by abstracting the concepts of users, extension numbers, and devices, and how to define the device configuration and authentication parameters in the channel configuration files. Next, we'll delve into the magic of Asterisk that is the dialplan, and see how simple things can create great results.

CHAPTER 6
Dialplan Basics

Everything should be made as simple as possible, but not simpler.
—Albert Einstein

The dialplan is the heart of your Asterisk system. It defines how calls flow into and out of the system. The dialplan is written in a scripting language, which specifies instructions that Asterisk follows in response to calls arriving from channels. In contrast to traditional phone systems, Asterisk's dialplan is fully customizable.

Experienced software developers find Asterisk dialplan code archaic, and often prefer to control call flow using Asterisk APIs such as AMI and ARI (which we will discuss in later chapters). Regardless of your plans in this regard, learning how Asterisk behaves is far easier if you understand dialplan first. It is perhaps also worth noting that Asterisk dialplan is performance-tuned, and is therefore the fastest way to execute call flow in terms of responsiveness and minimal load on the system. Dialplan is fast.

This chapter introduces the essential concepts of the dialplan, which will form the basis of any dialplan you write. Do not skip too much of this chapter, since the examples build on each other, and it is so fundamentally important to Asterisk. Please also note that this chapter is by no means an exhaustive survey of all the possible things dialplans can do; our aim is to cover just the essentials. We'll cover more advanced dialplan topics in later chapters. You are encouraged to experiment.

Dialplan Syntax

The Asterisk dialplan is specified in the configuration file named *extensions.conf*, located in the */etc/asterisk* directory.

The dialplan structure is built on four hierarchical components: Contexts, Extensions, Priorities, and Applications (see Figure 6-1).

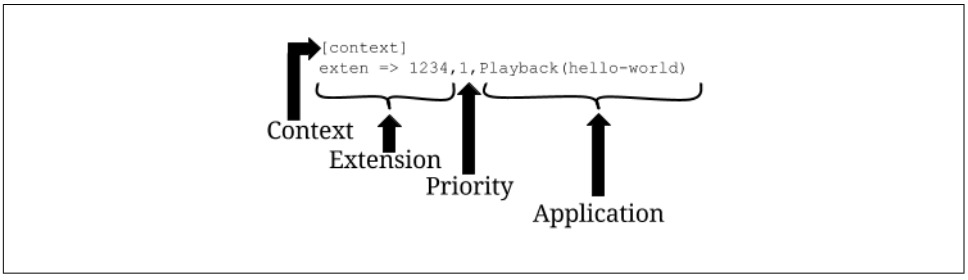

Figure 6-1. Dialplan hierarchy

Let's dive right in.

Sample Configuration Files

A basic *extensions.conf* file was created as part of the install process earlier in this book. We are going to build on that file in this chapter.

Asterisk also comes with a detailed *extensions.conf* file that can be installed with the sample configuration files (the installation command `make samples` will do this), and if you ran that command (which we don't recommend during the install, but is suggested by the installer), you will most likely have an */etc/asterisk/extensions.conf* file that is chock-full of information. Instead of starting with the sample file, we suggest that you build your *extensions.conf* file from scratch with a blank file (you can rename it or move it somewhere if you wish to keep it as a reference).

That being said, the *extensions.conf.sample* file is a fantastic resource, full of examples and ideas that you can use after you've learned the basic concepts. If you followed our installation instructions, you will find the file *extensions.conf.sample* in the folder *~/src/asterisk-15.<TAB>/configs/samples* (along with many other sample config files).

Contexts

Dialplans are divided into sections called *contexts*, which serve to separate different parts of the dialplan. An extension that is defined in one context is completely isolated from extensions in any other context, unless interaction is specifically allowed.

As a simple example, let's imagine we have two companies sharing an Asterisk server. If we place each company's automated attendant (IVR) in its own context, the two companies will be completely separated from each other. This allows us to independently define what happens when, say, extension 0 is dialed:

- Callers dialing 0 from Company A's voice menu need to be transferred to Company A's receptionist.
- Callers dialing 0 at Company B's voice menu will be sent to Company B's customer service department.

Both callers are on the same system, interacting with the same dialplan, but because they arrived in different contexts, they experience totally separate call flow. What happens to each incoming call is determined by the dialplan code in each context.

This is a very important consideration. With traditional PBXs, there is generally a set of defaults for things like reception, which means that if you forget to define them, they will probably work anyway. In Asterisk, the opposite is true. If you do not tell Asterisk how to handle every situation, and it comes across something it cannot handle, the call will typically be disconnected.

Contexts are defined in the *extensions.conf* file by placing the name of the context inside square brackets (`[]`). The name can be made up of the letters A through Z (upper- and lowercase), the numbers 0 through 9, and the hyphen and underscore.[1] A context for incoming calls from a carrier might simply be named this:

```
[incoming]
```

Context names have a maximum length of 79 characters (80 characters minus 1 terminating null).

Or perhaps:

```
[incoming_company_A]
```

Which then of course might require something like:

```
[incoming_company_B]
```

All of the instructions placed after a context definition are part of that context, until the next context is defined.

At the beginning of the dialplan, there are two special sections named `[general]` and `[globals]`. The `[general]` section contains a list of general dialplan settings (which you'll probably never have to worry about), and we will discuss the `[globals]`

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

context shortly. For now, it's only important to know that these two labels are not contexts, despite using context syntax. Do not use `[general]`, `[globals]`, and `[default]`[2] as context names, but otherwise name your contexts anything you wish.

The contexts in a typical *extensions.conf* file might be structured something like this:

```
[general] ; This always has to be here
[globals] ; Global variables (we'll discuss these later)

[incoming] ; Calls from the carriers could arrive here

[sets] ; on a simple system, we can use this

[sets1] ; Multi-tenanted perhaps needs this (sets from one company enter dialplan here)

[sets2] ; ... and this (another group of sets might enter the dialplan here)

[services] ; Special services such as conferencing could be defined here
```

When you define a channel (which is not done in the *extensions.conf* file), one of the required parameters in each channel definition is `context`. *The context is the point in the dialplan where connections coming from that channel will arrive.* So the way you plug a channel into the dialplan is through the context (see Figure 6-2).

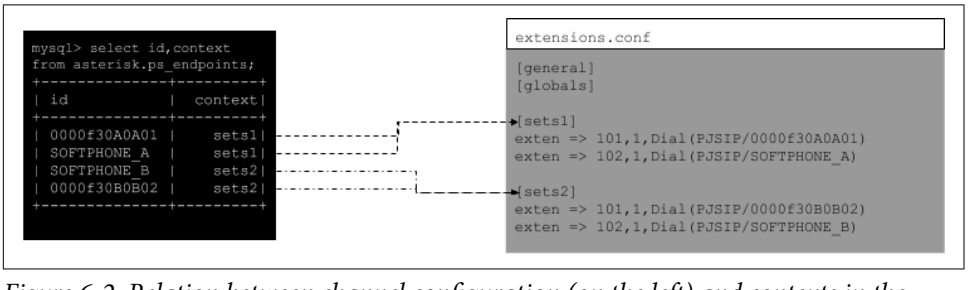

Figure 6-2. Relation between channel configuration (on the left) and contexts in the dialplan (on the right)

This is one of the most important concepts to understand when dealing with channels and dialplans. Troubleshooting call flow is much easier once you understand the relationship between the channel context (which tells the channel where to plug into the dialplan) and the dialplan context (which is where we create the call flow that happens when the call arrives).

An important (perhaps the most important) use of contexts is to provide privacy and security. By using contexts correctly, you can give some channels access to features

2 The default context used to be a popular way to whip up simple configurations, but this proved to be somewhat problematic for security. Best practice these days is to avoid all use of it.

(such as long-distance calling) that aren't made available to others. If you do not design your dialplan carefully, you may inadvertently allow others to fraudulently use your system. Please keep this in mind as you build your Asterisk system; there are many bots on the internet that were specifically written to identify and exploit poorly secured Asterisk systems.

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

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

Extensions

In the telecommunications industry the word *extension* typically has referred to a numeric identifier that, when dialed, will ring a phone (or system resource such as voicemail or a queue). In Asterisk, an extension is far more powerful, as it defines the unique series of steps (each step containing an application) through which Asterisk will take that call.

Within each context, we can define as many (or few) extensions as required. When a particular extension is triggered (by an incoming channel), Asterisk will follow the steps defined for that extension. It is the extensions, therefore, that specify what happens to calls as they make their way through the dialplan. Although extensions can, of course, be used to specify phone extensions in the traditional sense (i.e., extension 153 will cause the SIP telephone set on John's desk to ring), in an Asterisk dialplan, they can be used for much more.

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

```
exten =>
```

This is followed by the name (or number) of the extension.

When dealing with traditional telephone systems, we tend to think of extensions as the numbers you would dial to make another phone ring. In Asterisk, extension names can be any combination of numbers and letters. Over the course of this chapter and the next, we'll use both numeric and alphanumeric extensions.

Assigning names to extensions may seem like an unusual concept, but when you realize that SIP supports dialing by all sorts of character combinations (anything that is a valid URI, strictly speaking), it makes perfect sense. This is one of the features that makes Asterisk so flexible and powerful.

Each step in an extension has three components:

- The name (or number) of the extension
- The priority (each extension can include multiple steps; the step number is called the "priority")
- The application (or command) that will take place at that step

These three components are separated by commas, like this:

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

Here's a simple example:

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

The extension name is `123`, the priority is `1`, and the application is `Answer()`.

Priorities

Each extension can have multiple steps, called *priorities*. The priorities are numbered sequentially, starting with 1, and each executes one specific application. As an example, the following extension would answer the phone in priority number 1, and then hang it up in priority number 2. The steps in an extension take place one after the other.

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

It's pretty obvious that this code doesn't really do anything useful. We'll get there. The key point to note here is that for a particular extension, Asterisk follows the priorities in order.

```
exten => 123,1,Answer()
exten => 123,2,do something
exten => 123,3,do something else
exten => 123,4,do one last thing
exten => 123,5,Hangup()
```

This style of dialplan syntax is still seen from time to time, although (as you'll see momentarily) it is not generally used anymore for new code. Newer syntax is similar, but simplified.

Unnumbered priorities

In older releases of Asterisk, the numbering of priorities caused a lot of problems. Imagine having an extension that had 15 priorities, and then needing to add something at step 2: all of the subsequent priorities would have to be manually renumbered. Asterisk does not handle missing steps or misnumbered priorities, and debugging these types of errors was frustrating.

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

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

Internally, Asterisk will calculate the next priority number every time it encounters an `n`.[3] Now, if we want to add a new item at priority 3, we just input the new line where we need it, and no renumbering is required.

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

Bear in mind that you must always specify priority number 1. If you accidentally put an `n` instead of `1` for the first priority (a common mistake even among experienced dialplan coders), you'll find after reloading the dialplan that the extension will not exist.

The same => operator

In order to further simplify dialplan writing, a new syntax was created. As long as the extension remains the same, you can simply type `same =>` followed by the priority and application rather than having to type the full extension on each line:

3 Asterisk permits simple arithmetic within the priority, such as `n+200`, and the priority `s` (for same), but their usage is somewhat deprecated due to the existence of priority labels. Please note that *extension* `s` and *priority* `s` are two distinct concepts.

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

This style of dialplan will also make it easier to copy code from one extension to another. This is the preferred and recommended style. The only reason for the discussion of previous styles is to help understand how we got here.

> Make no mistake, the Asterisk dialplan is peculiar. Many folks avoid it altogether, and use AGI and ARI to write their dialplan.
>
> While there's certainly something to be said for writing dialplan in an external language (and we'll cover it in later chapters), the Asterisk dialplan is native to it, and you will not get better performance than this. Dialplan code executes fast.
>
> Also, if you want to understand how Asterisk thinks, you need to understand its dialplan.

Priority labels

Priority labels allow you to assign a name to a priority within an extension. This is to ensure that you can refer to a priority by something other than its number (which probably isn't known, given that dialplans now generally use unnumbered priorities). Later you will learn that it's often necessary to send calls from other parts of the dialplan to a particular priority in a particular extension. To assign a text label to a priority, simply add the label inside parentheses after the priority, like this:

```
exten => 123,n(label),application()
```

Later, we'll cover how to jump between different priorities based on dialplan logic. You'll see a lot more of priority labels, and you'll use them often in your dialplans.

A very common mistake when writing labels is to insert a comma between the `n` and the `(`, like this:

```
exten => 555,n,(label),application() ;<-- THIS WON'T WORK
exten => 556,n(label),application()  :<-- This is what we want
```

This mistake will break that part of your dialplan, and you will get an error stating that the application cannot be found.

Applications

Applications are the workhorses of the dialplan. Each application performs a specific action on the current channel, such as playing a sound, accepting touch-tone input,

looking something up in a database, dialing a channel, hanging up the call, feeding the cat, and so forth.[4] In the previous example, you were introduced to two simple applications: `Answer()` and `Hangup()`. It's obvious what they do, but it's also obvious that on their own they aren't terribly useful.

Some applications, including `Answer()` and `Hangup()`, need no other instructions to do their jobs. Most applications, however, require more information. These additional elements, or *arguments*, are passed on to the applications to affect how they perform their actions. To pass arguments to an application, place them between the parentheses that follow the application name, separated by commas.

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

The `Answer()` application is used to answer a channel that is ringing. It seems a simple thing, but a lot of things happen on the channel with this one command. `Answer()` tells the channel to send a message back to the far end that the call has been answered, and also to enable the media paths (the network streams that will carry the sound between the caller and the system). As we mentioned earlier, `Answer()` takes no arguments. `Answer()` is not always required (in fact, in some cases it may not be desirable at all), but it is an effective way to ensure a channel is connected before performing further actions.

The Progress() Application

Sometimes it is useful to be able to pass information back to the network before answering a call. The `Progress()` application attempts to provide call progress information to the originating channel. Some carriers expect this, and thus you may be able to resolve strange signaling problems by inserting `Progress()` into the dialplan where your incoming calls arrive. In terms of billing, the use of `Progress()` lets the carrier know you're handling the call, without starting the billing meter.

The `Playback()` application is used for playing a previously recorded sound file over a channel. Input from the user is ignored, which means that you would not use `Playback()` in an auto attendant, for example, unless you did not want to accept input at that point.[5]

4 OK, so feeding the cat isn't a common use for a telephone system, but through Asterisk, such things are not impossible. Doc Brown would've loved this thing.

5 There is another application called `Background()` that is very similar to `Playback()`, except that it does allow input from the caller. You can read more about this application in Chapters 14 and 16.

Asterisk comes with many professionally recorded sound files, which should be found in the default sounds directory (usually */var/lib/asterisk/sounds*). When you compile Asterisk, you can choose to install various sets of sample sounds that have been recorded in a variety of languages and file formats. We'll be using these files in many of our examples. Several of the files in our examples come from the Extra Sound Package, which we installed in Chapter 3. You can also have your own sound prompts recorded in the same voices as the stock prompts by visiting www.theivr-voice.com. Later in the book, we'll talk more about how you can use a telephone and the dialplan to create and manage your own system recordings (or import *.wav* files).

To use `Playback()`, specify a filename as the argument. For example, `Playback(filename)` would play a sound file called *filename.wav*, assuming it was located in the default sounds directory. Note that you can include the full path to a file if you want, like this:

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

The previous example would play *filename.wav* from the */home/john/sounds* directory. This can be problematic, however, due to potential file permissions problems. If you're planning on having a lot of custom sounds on your system, you'll likely want a dedicated directory for them, and you'll need to test to ensure Asterisk can find and play the files.

You can also use relative paths from the Asterisk sounds directory, as follows:

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

This example would play *filename.wav* from the *custom* subdirectory of the default sounds directory (probably */var/lib/asterisk/sounds/en/custom/filename.wav*). If the specified directory contains more than one file with that filename but with different file extensions, Asterisk automatically plays the best file.[6]

The `Hangup()` application does exactly as its name implies: it hangs up the active channel. You should use this application at the end of a context when you want to end the current call, to ensure that callers don't continue on in the dialplan in a way you might not have anticipated. The `Hangup()` application does not require any

6 Asterisk selects the best file based on translation cost—that is, it selects the file that is the least CPU-intensive to convert to its native audio format. When you start Asterisk, it calculates the translation costs between the different audio formats (they often vary from system to system). You can see these translation costs by typing **`core show translation`** at the Asterisk CLI. The numbers shown represent how many microseconds it takes Asterisk to transcode one second of audio.

arguments, but you can pass an ISDN cause code if you want, such as `Hangup(16)`, and it will be translated into a comparable SIP message and sent to the far end.

As we work through the book, we will be introducing you to many more Asterisk applications, but that's enough theory for now; let's write some dialplan!

A Basic Dialplan Prototype

To reiterate, then, the form of all dialplans is built from those four concepts: Context, Extension, Priority, and Application (Figure 6-3).

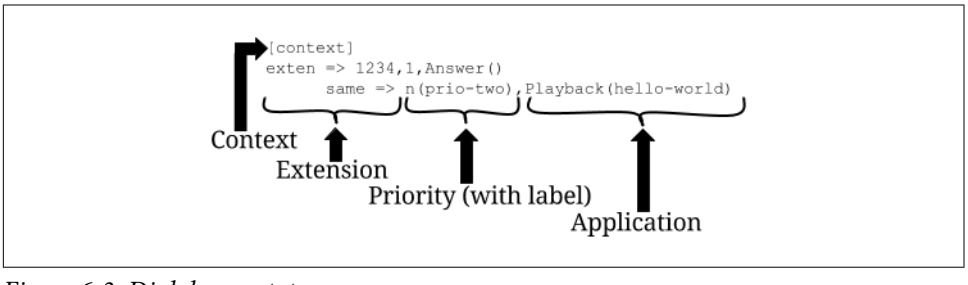

Figure 6-3. Dialplan prototype

A Simple Dialplan

OK, enough theory. Open up the file */etc/asterisk/extensions.conf* in your favorite editor, and let's take a look at your first dialplan (which was created in Chapter 5). We're going to add to that.

Hello World

As is typical in many technology books (especially computer programming books), our first example is called "Hello World."

In the first priority of our extension, we answer the call. In the second, we play a sound file named *hello-world*, and in the third we hang up the call. The code we are interested in for this example looks like this:

```
exten => 200,1,Answer()
     same => n,Playback(hello-world)
     same => n,Hangup()
```

If you followed along in Chapter 5, you'll already have a channel or two configured, as well as the sample dialplan that contains this code. If not, what you need is an *extensions.conf* file in your */etc/asterisk* directory that contains the following code:

```
[general]
[globals]

[sets]
```

```
exten => 100,1,Dial(PJSIP/0000f30A0A01) ; Replace 0000f30A0A01 with your device name

exten => 101,1,Dial(PJSIP/SOFTPHONE_A)

exten => 102,1,Dial(PJSIP/0000f30B0B02)

exten => 103,1,Dial(PJSIP/SOFTPHONE_B)

exten => 200,1,Answer()
     same => n,Playback(hello-world)
     same => n,Hangup()
```

If you don't have any channels configured, now is the time to do so. There is real satisfaction that comes from passing your first call into an Asterisk dialplan on a system that you've built from scratch. People get this funny grin on their faces as they realize that they have just created a telephone system. This pleasure can be yours as well, so please, don't go any further until you have made this little bit of dialplan work. If you have any problems, get back to Chapter 5 and work through the examples there.

If you don't have this dialplan code built yet, you'll need to add it and reload the dialplan with this CLI command:

```
$ sudo asterisk -rvvvvv # ('r' attaches to a daemonized Asterisk; 'v's are for verbosity)
*CLI> dialplan reload
```

or you can issue the command directly from the shell with:

```
$ sudo asterisk -rx "dialplan reload"  # ('rx' execute an Asterisk command and return)
```

Calling extension 200 from either of your configured phones[7] should reward you with the friendly voice of Allison Smith saying "Hello, World."

If it doesn't work, check the Asterisk console for error messages, and make sure your channels are assigned to the `sets` context.

We do not recommend that you move forward in this book until you have verified the following:

1. Calls between extension 100 and 101 are working.
2. Calling extension 200 plays "Hello World."

7 If you haven't configured two phones yet, please consider heading back to Chapter 5 and getting a couple of phones set up so you can play with them. You can get away with only one phone for testing, but really two is ideal. There are lots of free softphones available, and some of them are rather good.

Even though this example is very short and simple, it emphasizes the core dialplan concepts of contexts, extensions, priorities, and applications. You now have the fundamental knowledge on which all dialplans are built.

> As you build out a dialplan, it will be helpful to have the Asterisk CLI open in a new window. You will be reloading the dialplan often, and while testing your call flow, you will want to see what is happening, as it happens. The Asterisk CLI is useful for both of those things.
>
> ```
> $ sudo asterisk -rvvvvv
>
> *CLI> dialplan reload   # this Asterisk CLI command reloads the dialplan
> ```
>
> Best practice, then, would be to edit in one window, and to reload and debug in another.

Building an Interactive Dialplan

The dialplan we just built was static; it will always perform the same actions on every call. Many dialplans will also need logic to perform different actions based on input from the user, so let's take a look at that now.

The Goto(), Background(), and WaitExten() Applications

As its name implies, the `Goto()` application is used to send a call to another part of the dialplan. `Goto()` requires us to pass the destination context, extension, and priority as arguments, like this:

```
same => n,Goto(context,extension,priority)
```

We're going to create a new context called `TestMenu`, and create an extension in our `sets` context that will pass calls to that context using `Goto()`:

```
exten => 200,1,Answer()
     same => n,Playback(hello-world)
     same => n,Hangup()

exten => 201,1,Goto(TestMenu,start,1) ; add this to the end of the
                                      ; [sets] context

[TestMenu]
exten => start,1,Answer()
```

Now, whenever a device enters the `[sets]` context and dials 201, the call will be passed to the `start` extension in the `TestMenu` context (which currently won't do anything interesting because we still have more code to write).

We used the extension `start` in this example, but we could have used anything we wanted as an extension name, either numeric or alpha. We prefer to use alpha characters for extensions that are not directly dialable, as this makes the dialplan easier to read. Point being, we could have named our target extension `123` or `xyz321`, or `99luftballons`, or whatever we wanted instead of `start`. The word *start* doesn't mean anything special to the dialplan; it's simply the name of an extension.

One of the more useful applications in an interactive Asterisk dialplan is the `Background()`[8] application. Like `Playback()`, it plays a recorded sound file. Unlike `Playback()`, however, when the caller presses a key (or series of keys) on their telephone keypad, it interrupts the playback and passes the call to the extension that corresponds with the pressed digit(s). If a caller presses 5, for example, Asterisk will stop playing the sound prompt and send control of the call to the first priority of extension 5 (assuming there is an extension 5 to send the call to).

The most common use of the `Background()` application is to create basic voice menus (often called *auto attendants, IVRs*,[9] or *phone trees*). Many companies use voice menus to direct callers to the proper extensions, thus relieving their receptionists from having to answer every single call.

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

```
[TestMenu]
exten => start,1,Answer()
  same => n,Background(enter-ext-of-person)
```

If you want Asterisk to wait for input from the caller after the sound prompt has finished playing, you can use `WaitExten()`. The `WaitExten()` application waits for the caller to enter DTMF digits and is used directly following the `Background()` application, like this:

```
[TestMenu]
exten => start,1,Answer()
  same => n,Background(enter-ext-of-person)
  same => n,WaitExten()
```

8 It should be noted that some people expect that `Background()`, due to its name, will continue onward through the next steps in the dialplan while the sound is being played. In reality, its name refers to the fact that it is playing a sound in the background, while waiting for DTMF in the foreground.

9 More information about auto attendants and IVR can be found in Chapter 14.

If you'd like the `WaitExten()` application to wait a specific number of seconds for a response (instead of using the default timeout),[10] simply pass the number of seconds as the first argument to `WaitExten()`, like this:

```
same => n,WaitExten(5) ; We always pass a time argument to WaitExten()
```

Both `Background()` and `WaitExten()` allow the caller to enter DTMF digits. Asterisk then attempts to find an extension in the current context that matches the digits that the caller entered. If Asterisk finds a match, it will send the call to that extension. Let's demonstrate by adding a few lines to our dialplan example:

```
[TestMenu]
exten => start,1,Answer()
       same => n,Background(enter-ext-of-person)
       same => n,WaitExten(5)

exten => 1,1,Playback(digits/1)

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

After making these changes, save and reload your dialplan:

```
*CLI> dialplan reload
```

If you call into extension 201, you should hear a sound prompt that says, "Enter the extension of the person you are trying to reach." The system will then wait 5 seconds for you to enter a digit. If the digit you press is either 1 or 2, Asterisk will match the relevant extension, and read that digit back to you. Since we didn't provide any further instructions, your call will then end. You'll also find that if you enter a different digit (such as 3), the dialplan will be unable to proceed.

Let's embellish things a little. We're going to use the `Goto()` application to have the dialplan repeat the greeting after playing back the number:

```
[TestMenu]
exten => start,1,Answer()
       same => n,Background(enter-ext-of-person)
       same => n,WaitExten(5)

exten => 1,1,Playback(digits/1)
   same => n,Goto(TestMenu,start,1)

exten => 2,1,Playback(digits/2)
   same => n,Goto(TestMenu,start,1)
```

These new lines will send control of the call back to the `start` extension after playing back the selected number.

10 See the dialplan function `TIMEOUT()` for information on how to change the default timeouts. See Chapter 10 for information on what dialplan functions are.

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

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

Handling Invalid Entries and Timeouts

We need an extension for invalid entries. In Asterisk, when a context receives a request for an extension that is not valid within that context (e.g., pressing 9 in the preceding example), the call is sent to the `i` extension. We also need an extension to handle situations when the caller doesn't give input in time (the default timeout is 10 seconds). Calls will be sent to the `t` extension if the caller takes too long to press a digit after `WaitExten()` has been called. Here is what our dialplan will look like after we've added these two extensions:

```
[TestMenu]
exten => start,1,Answer()
       same => n,Background(enter-ext-of-person)
       same => n,WaitExten(5)

exten => 1,1,Playback(digits/1)
   same => n,Goto(TestMenu,start,1)

exten => 2,1,Playback(digits/2)
   same => n,Goto(TestMenu,start,1)

exten => i,1,Playback(pbx-invalid)
   same => n,Goto(TestMenu,start,1)

exten => t,1,Playback(please-try-again)
   same => n,Goto(TestMenu,start,1)
```

Using the `i`[11] and `t` extensions makes our menu a little more robust and user-friendly. That being said, it is still quite limited, because outside callers still have no way of connecting to a live person. To do that, we'll need to learn about the `Dial()` application.

11 The `i` extension is for catching invalid entries supplied to a dialplan application such as `Background()`. It is not used for matching on invalidly dialed extensions or nonmatching pattern matches.

Using the Dial() Application

One of Asterisk's most valuable features is its ability to connect different callers to each other. While Asterisk currently is used mostly for SIP connections, it supports a wide variety of channel types (from Analog to SS7, and various old VoIP protocols such as MGCP and SCCP). Asterisk takes much of the hard work out of connecting and translating between disparate networks. All you have to do is learn how to use the `Dial()` application.

The syntax of the `Dial()` application is more complex than that of the other applications we've used so far, but it's also where much of the magic of Asterisk happens. `Dial()` takes up to four arguments, which we'll look at next.

The syntax of `Dial()` looks like this:

```
Dial(Technology/Resource[&Technology2/Resource2[&...]][,timeout[,options[,URL]]])
```

Put simply, you tell `Dial()` what channel[12] you want to send the call out to, and set a few options to tweak the behavior. The use of `Dial()` can get complex, but at its most basic, it's that simple.

Argument 1: destination

The first argument is the destination you're attempting to call, which (in its simplest form) is made up of a technology (or transport) across which to make the call, a forward slash, and the address of the remote endpoint or resource.

These days, you're most likely to be using PJSIP as your channel type, but in the not-too-distant past, common technology types also included DAHDI (for analog and T1/E1/J1 channels), the old SIP channel (prior to PJSIP), and IAX2.[13] If you're looking at an older dialplan, you may see some of these other protocols represented. Going forward, only PJSIP and DAHDI are recommended and maintained.

Let's assume that we want to call one of our PJSIP channels named `SOFTPHONE_B`. The technology is `PJSIP`, and the resource (or channel) identifier is `SOFTPHONE_B`. Similarly, a call to a DAHDI device (defined in *chan_dahdi.conf*) might have a destination

12 Or channel**s**, if you want to ring more than one at a time.

13 IAX2 (pronounced "EEKS"), is the Inter Asterisk Exchange protocol (v2). In the early days of Asterisk it was popular for trunking, as it greatly reduced signaling overhead on busy circuits. Bandwidth has become far less expensive, and SIP protocol has become nearly ubiquitous. The IAX2 protocol is no longer actively maintained, but it still retains some popularity for its ability to traverse firewalls, and a few carriers might still support it. However, its use is deprecated, and in fact discouraged.

of DAHDI/14169671111. If we wanted Asterisk to ring the PJSIP/SOFTPHONE_B channel when extension 103 is reached in the dialplan, we'd add the following extension:

```
exten => 101,1,Dial(PJSIP/SOFTPHONE_A)

exten => 103,1,Dial(PJSIP/SOFTPHONE_B)

exten => 200,1,Answer()
```

We can also dial multiple channels at the same time, by concatenating the destinations with an ampersand (&), like this:

```
exten => 101,1,Dial(PJSIP/SOFTPHONE_A)

exten => 103,1,Dial(PJSIP/SOFTPHONE_B)

exten => 110,1,Dial(PJSIP/0000f30A0A01&PJSIP/SOFTPHONE_A&PJSIP/SOFTPHONE_B)

exten => 200,1,Answer()
```

The Dial() application will ring all of the specified destinations simultaneously, and bridge the inbound call with whichever destination channel answers first (the other channels will immediately stop ringing). If the Dial() application can't contact any of the destinations, Asterisk will set a variable called DIALSTATUS with the reason that it couldn't dial the destinations, and continue with the next priority in the extension.[14]

The Dial() application also allows you to connect to a remote VoIP endpoint not previously defined in one of the channel configuration files. The full syntax is:

```
Dial(technology/user[:password]@remote_host[:port][/remote_extension])
```

The full syntax for the Dial() application is slightly different for DAHDI channels:

```
Dial(DAHDI/[gGrR]channel_or_group[/remote_extension])
```

For example, here is how you would dial 1-800-555-1212 on DAHDI channel number 4:[15]

```
exten => 501,1,Dial(DAHDI/4/18005551212)
```

Argument 2: timeout

The second argument to the Dial() application is a timeout, specified in seconds. If a timeout is given, Dial() will attempt to call the specified destination(s) for that number of seconds before giving up and moving on to the next priority in the extension. If no timeout is specified, Dial() will continue to dial the called channel(s) until

14 We'll cover variables in the section "Using Variables" on page 96. In future chapters we'll discuss how to have your dialplan make decisions based on the value of DIALSTATUS.

15 Bear in mind that this assumes that this channel connects to something that knows how to reach external numbers.

someone answers or the caller hangs up. Let's add a timeout of 10 seconds to our extension:

```
exten => 101,1,Dial(PJSIP/SOFTPHONE_A)

exten => 102,1,Dial(PJSIP/0000f30B0B02,10)

exten => 103,1,Dial(PJSIP/SOFTPHONE_B)
```

If the call is answered before the timeout, the channels are bridged and the dialplan is done. If the destination simply does not answer, is busy, or is otherwise unavailable, Asterisk will set a variable called `DIALSTATUS` and then continue on with the next priority in the extension.

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

```
exten => 102,1,Dial(PJSIP/0000f30B0B02,10)
   same => n,Playback(vm-nobodyavail)
   same => n,Hangup()
```

As you can see, this example will play the *vm-nobodyavail.gsm* sound file if the call goes unanswered (and then hang up). Note that this doesn't actually provide voicemail; we're just playing a prompt, which could have been any valid prompt. We'll cover sending calls to voicemail later.

Argument 3: option

The third argument to `Dial()` is an option string. It may contain one or more characters that modify the behavior of the `Dial()` application. While the list of possible options is too long to cover here, one of the most popular is the `m` option. If you place the letter `m` as the third argument, the calling party will hear hold music instead of ringing while the destination channel is being called (assuming, of course, that music on hold has been configured correctly). To add the `m` option to our last example, we simply change the first line:

```
exten => 102,1,Dial(PJSIP/0000f30B0B02,10,m)
  same => n,Playback(vm-nobodyavail)
  same => n,Hangup()
```

Argument 4: URI

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

Updating the dialplan

Let's modify extensions 1 and 2 in our menu to use the `Dial()` application, and add extensions 3 and 4 just for good measure:

```
[TestMenu]
exten => start,1,Answer()
   same => n,Background(enter-ext-of-person)
   same => n,WaitExten(5)

exten => 1,1,Dial(PJSIP/0000f30A0A01,10)
   same => n,Playback(vm-nobodyavail)
   same => n,Hangup()

exten => 2,1,Dial(PJSIP/0000f30B0B02,10)
   same => n,Playback(vm-nobodyavail)
   same => n,Hangup()

exten => 3,1,Dial(PJSIP/SOFTPHONE_A,10)
   same => n,Playback(vm-nobodyavail)
   same => n,Hangup()

exten => 4,1,Dial(PJSIP/SOFTPHONE_B,10)
   same => n,Playback(vm-nobodyavail)
   same => n,Hangup()

exten => i,1,Playback(pbx-invalid)
   same => n,Goto(TestMenu,start,1)

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

Blank arguments

Note that the second, third, and fourth arguments may be left blank; only the first argument is required. For example, if you want to specify an option but not a timeout, simply leave the timeout argument blank, like this:

```
exten => 4,1,Dial(SIP/SOFTPHONE_B,,m)
```

Using Variables

If you have programming experience, you already understand what a variable is. If not, we'll briefly explain what variables are and how they are used. Any dialplan work beyond the very simple examples just given will greatly benefit from the use of variables. They are one of the useful features of a customizable dialplan that you will not find in a typical proprietary PBX.

A *variable* is a named container that can hold a value. Think of it like a post office box. The advantage of a variable is that its contents may change, but its name does not, which means you can write code that references the variable name and not worry about what the value will be. It is almost impossible to do any sort of useful programming without variables.

There are two ways to reference a variable. To reference the variable's name, simply type the name of the variable. If, on the other hand, you want to reference the value of the variable, you must type a dollar sign, an opening curly brace, the name of the variable, and a closing curly brace. So, to use the post office box analogy, you refer to the box itself by simply using its name, and you refer to the contents with the use of the `${}` wrapper. A variable named MyVar is referred to as `MyVar`, and its contents are accessed with `${MyVar}`. Here's how we might use a variable inside the `Dial()` application:[16]

```
exten => 203,1,Noop(say some digits)
  same => n,Answer()
  same => n,Set(SomeDigits=123)
  same => n,SayDigits(${SomeDigits})
  same => n,Wait(.25)
  same => n,Set(SomeDigits=543)
  same => n,SayDigits(${SomeDigits})
```

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

Note that variable names are case-sensitive. A variable named `SOME DIGITS` is different from a variable named `SomeDigits`. You should also be aware that any variables set by Asterisk will be uppercase. Some variables, such as `CHANNEL` and `EXTEN`, are reserved by Asterisk. You should not attempt to set these variables. It is popular to write global variables in uppercase and channel variables in Pascal/Camel case, but it is not strictly required.

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

Global variables

As their name implies, *global* variables are visible to all channels at all times. Global variables are useful in that they can be used anywhere within a dialplan to increase readability and manageability. Suppose for a moment that you had a large dialplan and several hundred references to the `PJSIP/0000f30A0A01` channel. Now imagine you replaced the phone with a different unit (perhaps a different MAC address), and had to go through your dialplan and change all of those references to `PJSIP/0000f30A0A01`. Not pretty.

On the other hand, if you had defined a global variable that contained the value `PJSIP/0000f30A0A01` at the beginning of your dialplan and then referenced that

16 Specifically, what we are setting here is a channel variable.

instead, you would have to change only one line of code to affect all places in the dialplan where that channel was used.

Global variables should be declared in the `[globals]` context at the beginning of the *extensions.conf* file. As an example, we will create a few global variables that store the channel identifiers of our devices. These variables are set at the time Asterisk parses the dialplan:

```
[globals]
UserA_DeskPhone=PJSIP/0000f30A0A01
UserA_SoftPhone=PJSIP/SOFTPHONE_A
UserB_DeskPhone=PJSIP/0000f30B0B02
UserB_SoftPhone=PJSIP/SOFTPHONE_B
```

We'll come back to these later.

Channel variables

A *channel* variable is a variable that is associated only with a particular call. Unlike global variables, channel variables are defined only for the duration of the current call and are available only to the channels participating in that call.

There are many predefined channel variables available for use within the dialplan, which are explained in the Asterisk wiki. You define a channel variable with extension 203 and the `Set()` application:

```
exten => 203,1,Noop(say some digits)
  same => n,Set(SomeDigits=123)
  same => n,SayDigits(${SomeDigits})
  same => n,Wait(.25)
  same => n,Set(SomeDigits=543)
  same => n,SayDigits(${SomeDigits})
```

You're going to be seeing a lot more channel variables. Read on.

Environment variables

Environment variables are a way of accessing Unix environment variables from within Asterisk. These are referenced using the `ENV()` dialplan function.[17] The syntax looks like `${ENV(`*var*`)}`, where *var* is the Unix environment variable you wish to reference. Environment variables aren't commonly used in Asterisk dialplans, but they are available should you need them.

17 We'll get into dialplan functions later. Don't worry too much about environment variables right now. They are not important to understanding the dialplan.

Adding variables to our dialplan

Now that we've learned about variables, let's put them to work in our dialplan. We're going to add three global variables that will associate a variable name to a channel name:

```
[general]
[globals]
UserA_DeskPhone=PJSIP/0000f30A0A01
UserA_SoftPhone=PJSIP/SOFTPHONE_A
UserB_DeskPhone=PJSIP/0000f30B0B02
UserB_SoftPhone=PJSIP/SOFTPHONE_B

[sets]
exten => 100,1,Dial(${UserA_DeskPhone})

exten => 101,1,Dial(${UserA_SoftPhone})

exten => 102,1,Dial(${UserB_DeskPhone},10)
  same => n,Playback(vm-nobodyavail)
  same => n,Hangup()

exten => 103,1,Dial(${UserB_SoftPhone})

exten => 110,1,Dial(${UserA_DeskPhone}&${UserA_SoftPhone}&${UserB_SoftPhone})

exten => 200,1,Answer()
```

Let's update the test menu as well:

```
[TestMenu]
exten => start,1,Answer()
   same => n,Background(enter-ext-of-person)
   same => n,WaitExten(5)

exten => 1,1,Dial(${UserA_DeskPhone},10)
   same => n,Playback(vm-nobodyavail)
   same => n,Hangup()

exten => 2,1,Dial(${UserA_SoftPhone},10)
   same => n,Playback(vm-nobodyavail)
   same => n,Hangup()

exten => 3,1,Dial(${UserB_DeskPhone},10)
   same => n,Playback(vm-nobodyavail)
   same => n,Hangup()

exten => 4,1,Dial(${UserB_SoftPhone},10)
   same => n,Playback(vm-nobodyavail)
   same => n,Hangup()

exten => i,1,Playback(pbx-invalid)
```

It rarely makes sense to hardcode data in a dialplan. It's almost always better to use a variable.

Make sure you test this out to ensure you don't have any typos, and also to see what it looks like when executed on the Asterisk CLI:

```
# asterisk -rvvvvvv
*CLI> dialplan reload
    -- Executing [201@sets:1] Goto("PJSIP/0000f30A0A01", "TestMenu,start,1")
    -- Goto (TestMenu,start,1)
    -- Exec [start@TestMenu:1] Answer("PJSIP/0000f30A0A01", "")
    -- Exec [start@TestMenu:2] BackGround("PJSIP/0000f30A0A01", "enter-ext-of-person")
    -- <PJSIP/0000f30A0A01> Playing 'enter-ext-of-person.slin' (language 'en')
    -- Exec [1@TestMenu:1] Dial("PJSIP/0000f30A0A01", "PJSIP/0000f30A0A01,10")
    -- Called PJSIP/0000f30A0A01
    -- PJSIP/0000f30A0A01-00000011 is ringing
  == Spawn extension (TestMenu, 1, 1) exited non-zero on 'PJSIP/0000f30A0A01'
```

Variable concatenation

To concatenate variables, simply place them together, like this:

```
exten => 204,1,Answer()
  same => n,Answer()
  same => n,Set(ONETWO=12)
  same => n,Set(THREEFOUR=34)
  same => n,SayDigits(${ONETWO}${THREEFOUR})   ; easy peasy
  same => n,Wait(0.2)
  same => n,Set(NOTFIVE=${THREEFOUR}${ONETWO}) ; peasy easy
  same => n,SayNumber(${NOTFIVE})              ; see what we did here?
  same => n,Wait(0.2)
  same => n,SayDigits(2${ONETWO}3)             ; you can concatenate literals and variables
```

Inheriting channel variables

Channel variables are always associated with the original channel that set them, and are no longer available once the channel is transferred.

In order to allow channel variables to follow the channel as it is transferred around the system, you must employ channel variable inheritance. There are two modifiers that can allow the channel variable to follow the channel: single underscore and double underscore.

The single underscore (_) causes the channel variable to be inherited by the channel for a single transfer, after which it is no longer available for additional transfers. If you use a double underscore (__), the channel variable will be inherited throughout the life of that channel.

Setting channel variables for inheritance simply requires you to prefix the channel name with a single or double underscore. The channel variables are then referenced exactly the same as they would be normally.

Here's an example of setting a channel variable for single transfer inheritance:

```
exten => example,1,Set(_MyVariable=thisValue)
```

Here's an example of setting a channel variable for infinite transfer inheritance:

```
exten => example,1,Set(__MyVariable=thisValue)
```

When you wish to read the value of the channel variable, you do not use the underscore(s):

```
exten => example,1,Verbose(1,Value of MyVariable is: ${MyVariable})
```

Pattern Matching

If we want to be able to allow people to dial *through* Asterisk and have Asterisk connect them to outside resources, we need a way to match on any possible phone number that the caller might dial. For situations like this, Asterisk offers *pattern matching*. Pattern matching allows you to create one extension in your dialplan that matches many different numbers. This is enormously useful.

Pattern-matching syntax

When we are using pattern matching, certain letters and symbols represent what we are trying to match. *Patterns always start with an underscore* (_). This tells Asterisk that we're matching on a pattern, and not on an explicit extension name.

If you forget the underscore at the beginning of your pattern, Asterisk will think it's just a named extension and won't do any pattern matching. This is one of the most common mistakes people make when starting to learn Asterisk.

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

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

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

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

Another common mistake is to try to use the letters `X`, `Z`, and `N` literally in a pattern match; to do that, wrap them in square brackets (case-insensitive), such as `_ale[X][Z]a[N]der`.

`[15-7]`
: Matches a single character from the range of digits specified. In this case, the pattern matches a single 1, as well as any number in the range 5, 6, 7.

`.` *(period)*
: Wildcard match; matches *one or more* characters, no matter what they are.

If you're not careful, wildcard matches can make your dialplans do things you're not expecting (like matching built-in extensions such as `i` or `h`). You should use the wildcard match in a pattern only after you've matched as many other digits as possible. For example, the following pattern match should probably never be used:

```
_.
```

In fact, Asterisk will warn you if you try to use it. Instead, if you really need a catchall pattern match, use this one to match all strings that start with a digit followed by one or more characters (see `!` if you want to be able to match on zero or more characters):

```
_X.
```

Or this one, to match any alphanumeric string:

```
_[0-9a-zA-Z].
```

! *(bang)*
: Wildcard match; matches *zero or more* characters, no matter what they are.

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

```
exten => _4XX,1,Noop(User Dialed ${EXTEN})
  same => n,Answer()
  same => n,SayDigits(${EXTEN})
  same => n,Hangup()
```

In this example, the pattern matches any three-digit extension from 400 through 499.[18]

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

```
exten => _555XXXX,1,Answer()
  same => n,SayDigits(${EXTEN})
exten => _55512XX,1,Answer()
  same => n,Playback(tt-monkeys)
```

18 We've used the `EXTEN` channel variable without introducing it. Read on, as it will be covered later in this chapter.

In this case the second extension would be selected, because it is more specific. Load this in and make calls to 5550000, 5550123, 5551212, 5551200, 5551300, 5551299, and so forth to get a feel for how this works. Play around with different pattern matches. For example, what would pattern `_555NNNN` match? What would pattern `_[0-9].` match?

North American Numbering Plan—pattern-matching examples

This pattern matches any seven-digit number, as long as the first digit is 2 or higher:

```
_NXXXXXX
```

The preceding pattern would be compatible with any North American Numbering Plan local seven-digit number.

In areas with 10-digit dialing, that pattern would look like this:

```
_NXXNXXXXXX
```

Note that neither of these two patterns would handle long-distance calls. We'll cover those shortly.

The NANP and Toll Fraud

The North American Numbering Plan (NANP) is a shared telephone numbering scheme used by 19 countries in North America and the Caribbean. All of these countries share country code 1.

In the United States and Canada, there is sufficient competition that you can place a long-distance call to most numbers in country code 1 and expect to pay a reasonable toll. However, many people don't realize that 17 other countries, many of which have very different telecom regulations, share the NANP. Some of these places are quite expensive to call.

One popular scam using the NANP tries to trick naïve North Americans into calling expensive per-minute toll numbers in a Caribbean country; the callers believe that since they dialed 1-NPA-NXX-XXXX to reach the number, they'll be paying their standard national long-distance rate for the call. Since the country in question may have regulations that allow for this form of extortion, the caller is ultimately held responsible for the call charges.

It may be prudent to block calls to area codes to NANP countries outside the US and Canada until you've had a chance to review your toll rates to those countries. Wikipedia has a good reference for the basics of what you need to know about NANP, including what NPAs (area codes) belong to what country.

Let's try another:

```
_1NXXNXXXXXX
```

This one will match the number 1, followed by an area code between 200 and 999, then any seven-digit number that does not start with 0 or 1. In the NANP calling area, you would use this pattern to match any long-distance number.[19]

And finally this one:

```
_011.
```

Note the period on the end. This pattern matches any number that starts with 011 and has at least one more digit. In the NANP, this indicates an international phone number. (We'll be using these patterns in the next section to add outbound dialing capabilities to our dialplan.)

Common global pattern matches

Outside of North America, there is wide variance in how numbering is handled; however, some patterns are common. Here are a few simple examples:

```
; UK, Germany, Italy, China, etc.
exten => _00X.,1,noop()   ; international dialing code
exten => _0X.,1,noop()    ; national dialing prefix
exten => 112,1,Noop(--==[ Emergency call ]==--)

; Australia
exten => _0011X.,1,noop() ; international dialing code
exten => _0X.,1,noop()    ; national dialing prefix

; Dutch Caribbean (Saba)
exten => _00X.,1,noop()        ; international
exten => _416XXXX,1,noop()     ; local (on-island)
exten => _0[37]XXXXXX,1,noop() ; call to country code 599 off-island (not Curacao)
exten => _09XXXXXXX,1,Noop()   ; call to country code 599 off-island (Curacao)
```

You will need to understand the dialing plan of your region in order to produce a useful pattern match.

Using the ${EXTEN} channel variable

So what happens if you want to use pattern matching but need to know which digits were actually dialed? Enter the `${EXTEN}` channel variable. Whenever you dial an extension, Asterisk sets the `${EXTEN}` channel variable to the digits that were received. We used the application `SayDigits()` to demonstrate this.

19 If you grew up in North America, you may believe that the 1 you dial before a long-distance call is "the long-distance code." This is not completely correct. The number 1 is also the international country code for NANP. Keep this in mind if you send your phone number to someone in another country. The recipient may not know your country code, and thus be unable to call you with just your area code and phone number. Your full phone number with country code is +1 NPA NXX XXXX (where NPA is your area code)—for example, +1 416 555 1212. This is also known as E.164 format (Wikipedia can tell you all about E.164).

```
exten => _4XX,1,Noop(User Dialed ${EXTEN})
  same => n,Answer()
  same => n,SayDigits(${EXTEN})
  same => n,Hangup()

exten => _555XXXX,1,Answer()
  same => n,SayDigits(${EXTEN})
```

In these examples, the `SayDigits()` application read back to you the extension you dialed.

Often, it's useful to manipulate the `${EXTEN}` by stripping a certain number of digits off the front of the extension. This is accomplished by using the syntax `${EXTEN:`*x*`}`, where *x* is where you want the returned string to start, from left to right. For example, if the value of `${EXTEN}` is `95551212`, `${EXTEN:1}` equals `5551212`. Let's try another example:

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

In this example, the `SayDigits()` application would start at the second digit, and thus read back only the last two digits of the dialed extension.

More Advanced Digit Manipulation

The `${EXTEN}` variable properly has the syntax `${EXTEN:`*x*`:`*y*`}`, where *x* is the starting position and *y* is the number of digits to return. Given the following dial string:

```
94169671111
```

we can extract the following digit strings using the `${EXTEN:`*x*`:`*y*`}` construct:

- `${EXTEN:1:3}` would contain `416`
- `${EXTEN:4:7}` would contain `9671111`
- `${EXTEN:-4:4}` would start four digits from the end and return four digits, giving us `1111`
- `${EXTEN:2:-4}` would start two digits in and exclude the last four digits, giving us `16967`
- `${EXTEN:-6:-4}` would start six digits from the end and exclude the last four digits, giving us `67`
- `${EXTEN:1}` would give us everything after the first digit, or `4169671111` (if the number of digits to return is left blank, it will return the entire remaining string)

This is a very powerful construct, but most of these variations are not very common in normal use. For the most part, you will be using `${EXTEN}` (or perhaps `${EXTEN:1}` if you need to strip off an external access code, such as a prepended 9).

Includes

Asterisk has an important feature that allows extensions from one context to be available from within another context. This is accomplished through use of the `include` directive, which allows us to control access to different sections of the dialplan.

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

```
include => context
```

Including one context within another context allows extensions within the included context to be dialable.

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

We will discuss the `include` directive more in Chapter 7.

Conclusion

And there you have it—a basic but functional dialplan. There is still much we have not covered, but you've got all of the fundamentals. In the following chapters, we'll continue to build on this foundation.

If parts of this dialplan don't make sense, you may want to go back and reread a section or two before continuing on to the next chapter. It's imperative that you understand these principles and how to apply them, as the next chapters build on this information.

CHAPTER 7
Outside Connectivity

You cannot always control what goes on outside. But you can always control what goes on inside.

—Wayne Dyer

In the previous chapters, we have covered a lot of important information that is essential to a working Asterisk system. However, we have yet to discuss something that is vital to any useful PBX: namely, connecting it to the outside world. In this chapter we will discuss outside connectivity.

The groundbreaking architecture of Asterisk was significant due in large part to the fact that it treats all channel types as equal. This is in contrast to a traditional PBX, where trunks (which connect to the outside world) and extensions (which connect to users and resources) are logically separated. The fact that the Asterisk dialplan treats all channels in a similar manner means that in an Asterisk system you can accomplish very easily things that are much more difficult (or impossible) to achieve on a traditional PBX.

This flexibility does come with a price, however. Since the system does not inherently know the difference between an internal resource (such as a telephone set) and an external resource (such as a telco circuit), it is up to you to ensure that your dialplan handles each type of resource appropriately.

The Basics of Trunking

The purpose of *trunking* is to provide a shared connection between two entities. Trees have trunks, and everything that passes between the roots and the leaves happens through the trunk. Railroads use the term "trunk" to refer to a major line that connects feeder lines together.

In telecommunications, trunking connects two systems together. Carriers use trunks to connect their networks to each other. In a PBX, the circuits that connect the PBX to the outside world are (from the perspective of the PBX) usually referred to as trunks (although the carriers themselves do not generally consider these to be trunks). From a technical perspective, the definition of a trunk is not as clear as it used to be (PBX trunks used totally different technology from station circuits, but now both are usually SIP), but as a concept, trunks are still important. With SIP, everything is technically peer-to-peer, so from a technology perspective there isn't really such a thing as a trunk anymore (or perhaps it's more accurate to say that everything is a trunk). From a functional perspective it is still useful to be able to differentiate between VoIP resources that connect to the outside world (trunks, lines, circuits, etc.) and VoIP resources that connect to user endpoints (stations, sets, extensions, handsets, telephones, etc.).

In an Asterisk PBX, you might have trunks that go to your VoIP provider for in-country long-distance calls, trunks for your overseas calls, and trunks that connect your various offices together. These trunks might actually run across the same network connection, but in your dialplan you could treat them quite differently. You can even have a trunk in Asterisk that simply loops back in on itself (which is usually some kludgy hack that solves some namespace or CDR problem that wasn't getting solved any other way).

Fundamental Dialplan for Outside Connectivity

In a traditional PBX, external lines are generally accessed by way of an access code that must be dialed before the number.[1] It is common to use the digit 9 for this purpose.

In Asterisk, it is similarly possible to assign 9 for routing of external calls, but since the Asterisk dialplan is so much more intelligent, it is not really necessary to force your users to dial 9 before placing a call. Typically, you will have an extension range for your system (say, 100–199), and a feature code range (*00 to *99). Anything outside those ranges that matches the dialing pattern for your country or region can be treated as an external call.

If you have one carrier providing all of your external routing, you can handle your external dialing through a few simple pattern matches. The example in this section is valid for the North American Numbering Plan (NANP). If your country is not within the NANP (which serves Canada, the US, and many Caribbean countries), you will need a different pattern match.

1 In a key system, each line has a corresponding button on each telephone, and lines are accessed by pressing the desired line key.

The `[globals]` section contains two variables, named `LOCAL` and `TOLL`.[2] The purpose of these variables is to simplify management of your dialplan should you ever need to change carriers. They allow you to make one change to the dialplan that will affect all places where the specified channel is referenced:

```
[globals]
; These channels are the same to asterisk as
; any PJSIP enpoint, so they'll be configured
; similar to telephone sets.
; Each carrier will have their own configuration
; requirements (although they'll all be similar)
LOCAL=PJSIP/my-itsp
TOLL=PJSIP/my-other-itsp
```

The `[external]` section contains the actual dialplan code that will recognize the numbers dialed and pass them to the `Dial()` application:[3]

```
[external]
exten => _NXXNXXXXXX,1,Dial(${LOCAL}/${EXTEN}) ; 10-digit pattern match for NANP
exten => _NXXXXXX,1,Dial(${LOCAL}/${EXTEN})    ; 7-digit pattern match for NANP
exten => _1NXXNXXXXXX,1,Dial(${TOLL}/${EXTEN}) ; Long-distance pattern match
                                                ; for NANP
exten => _011.,1,Dial(${TOLL}/${EXTEN})        ; International pattern match for
                                                ; calls made from NANP

; This section is functionally the same as the above section.
; It is for people who like to dial '9' before their calls
exten => _9NXXNXXXXXX,1,Dial(${LOCAL}/${EXTEN:1})
exten => _9NXXXXXX,1,Dial(${LOCAL}/${EXTEN:1})
exten => _91NXXNXXXXXX,1,Dial(${TOLL}/${EXTEN:1})
exten => _9011.,1,Dial(${TOLL}/${EXTEN:1})
```

In any context that would be used by sets or user devices, you would use an `include => ` directive to allow access to the `external` context:

```
[sets]
include => external
```

It is critically important that you do not include access to the external lines in any context that might process an incoming call. The risk here is that a phishing bot could eventually gain access to your outgoing trunks (you'd be surprised at how common these phishing bots are).

We cannot stress enough how important it is that you ensure that no external resource can access your toll lines.

2 You can name these anything you wish.

3 For more information on pattern matches, see Chapter 6.

The PSTN

The public switched telephone network (PSTN) has existed for over a century. It is the precursor to many of the technologies that shape our world today, from the internet to MP3 players.

The use of old-school PSTN circuits in Asterisk systems is no longer common. The technical complexities, costs, and limitations of obsolete technology are only justified in situations where a reliable internet connection is not available (and even then, traditional circuits will often be a problematic choice). Even the carriers themselves have largely switched to VoIP for their internal backbones.

The PSTN Has Retired

More than any technical factor, perhaps the most significant nail in the PSTNs coffin is the fact that most of the technical experts in the field of traditional telephony are near or past retirement age, and the new kids have no interest in this sort of thing. Point being: you will increasingly find that carriers no longer have the skilled staff required to deploy traditional PSTN services. All the cool kids are learning VoIP (which is ultimately just a networking technology), and all the carriers put their best and brightest on the VoIP/SIP side of the business.

So, while it used to be true that you couldn't beat a PRI circuit for reliability, that is no longer the case. In fact, many companies deliver PRI circuits across a SIP connection, which is a kludge Asterisk has no use for.

Where the PSTN might still hold sway for a few years more is in telephone numbers. If VoIP had been invented without the PSTN preceding it, it's unlikely that something like a phone number would have ever been invented. Still, we've got them, and we use them, and the reason we do so is perhaps not so much due to any usefulness they provide, but rather due to the fact that they are managed by a complex, multinational consortium of standards bodies and curators who ensure the integrity of the global call routing plan.

To put the value of this in perspective, it might be worth considering that if the internet had designed the telephone network (and phone calls were as free as email), all our SIP phones would likely be ringing all day long with one spammy call after another. That still happens, but it's greatly reduced by the fact that a phone call costs money, and even if it costs just pennies, that's enough to keep much of the exceedingly mindless spam out of the game.

Another feature the PSTN offers is standards compliance and interoperability. If you look at any internet-based voice product, they are either proprietary walled gardens, or they are community-driven and have failed to gain any useful traction. It is our

belief that this will not change until some sort of trust mechanism exists that ensures the identities of incoming callers have been verified by some widely recognized authority.

Traditional PSTN Trunks

This section has been written as a nod to the telecommunications industry, and to the history of Asterisk itself. It is in part because Asterisk could talk to so many different sorts of old-school circuits that it achieved the early success it did. These days, the use of these old circuits has for the most part faded into history.

There are two types of fundamental technology that PSTN carriers have used to deliver telephone circuits: analog and digital.

Analog telephony

The first telephone networks were all analog. The audio signal that you generated with your voice was used to generate an electrical signal, which was carried to the other end. The electrical signal had the same characteristics as the sound being produced.

Analog circuits have several characteristics that differentiate them from other circuits you might wish to connect to Asterisk:

- No signaling channel exists—line state signaling is electromechanical, and addressing is done using in-band audio tones.
- Disconnect supervision is usually delayed by several seconds, and is not completely reliable.
- Far-end supervision is minimal (for example, answer supervision is lacking).
- Differences in circuits mean that audio characteristics will vary from circuit to circuit and will require tuning.

Incoming analog circuits that you wish to connect to your Asterisk system will need to connect to a Foreign eXchange Office (FXO) port. Since there is no such thing as an FXO port in any standard computer, an FXO port of some sort must be provided to the system before you can connect traditional analog lines. Companies such as Digium and Sangoma offer such cards, but you can also purchase a SIP device that provides such ports.

FXO and FXS

For any analog circuit, there are two ends: the office (typically the central office of the PSTN) and the station (typically a phone, but could also be a card such as a modem or line card in a PBX).

The central office is responsible for:

- Power on the line (nominally 48 volts DC)
- Ringing voltage (nominally 90 volts AC)
- Providing dialtone
- Detecting hook state (off-hook and on-hook)
- Sending supplementary signaling such as caller ID

The station is responsible for:

- Providing a ringer (or at least being able to handle ringing voltage in some manner)
- Providing a dialpad (or some way of sending DTMF)
- Providing a hook switch to indicate the status of the line

A Foreign eXchange (FX) port *is named by what it connects to*, not by what it does. So, for example, a Foreign eXchange Office (FXO) port is actually a station: it connects to the central office. A Foreign eXchange Station (FXS) port is actually a port that provides the services of a central office (in other words, you would plug an analog set into an FXS port).

Note that changing from FXO to FXS is not something you can simply do with a settings change. FXO and FXS ports require completely different electronics.

This stuff is old-school, folks. You can run old phones from 100 years ago off an FXS port!

We do not recommend the use of analog trunks in an Asterisk system. Their configuration and use is outside of the scope of this book.[4]

4 We should note, however, that we've written extensively on the subject in the past, and that body of work has been released under Creative Commons licensing, and is freely available.

Digital telephony

Digital telephony was developed in order to overcome many of the limitations of analog. Some of the benefits of digital circuits include:

- No loss of amplitude over long distances
- Reduced noise on circuits (especially long-distance circuits)
- Ability to carry more than one call per physical circuit
- Faster call setup and teardown
- Richer signaling information (especially if using ISDN)
- Lower cost for carriers
- Lower cost for customers (at higher densities)

There were several fundamental digital circuits that gained wide acceptance in the telecommunications industry:

T1 (24 channels)
: Used in Canada and the United States (mostly for ISDN-PRI)[5]

E1 (32 channels)
: Used in the rest of the world (ISDN-PRI or MFC/R2)

BRI (2 channels)
: Used for ISDN-BRI circuits (Euro-ISDN)

Note that the physical circuit can be further defined by the protocol running on the circuit. For example, a T1 could be used for either ISDN-PRI or CAS, and an E1 could be used for ISDN-PRI, CAS, or MFC/R2.

It is difficult to justify these circuit types anymore. Relative to VoIP protocols, they have become expensive, complex, and somewhat inflexible. If you need to connect such circuits to an Asterisk system, we recommend some sort of gateway device to convert the circuit into SIP, and then connect via SIP to your Asterisk system. If you want a single-chassis system, companies such as Digium and Sangoma offer digital PSTN cards that can be installed directly into your Asterisk server; they are

5 There's also a circuit used in Japan called a J-1, which is most simply described as a 24-channel E1.

connected to Asterisk by way of the DAHDI channel driver. The use of this technology is outside of the scope of this book.[6]

VoIP

Compared to the lengthy history of the telecommunications industry,[7] VoIP is still a relatively new concept. For the century or so prior to VoIP, the only way to connect your site to the PSTN was through the use of circuits provided for that purpose by your local telephone company. VoIP now allows for connections between endpoints without the PSTN having to be involved at all (although in most VoIP scenarios, there will still be a PSTN component at some point, especially if there is a traditional E.164 phone number involved). The PSTN still controls the phone numbers, and we'll be using those until somebody comes up with an internet-based addressing mechanism that is not subject to abuse the way email has been.[8]

Network Address Translation

If you are going to be using VoIP across any sort of wide-area network (such as the internet), you will be dealing with firewalls, and quite frequently network address translation (NAT) as well.[9] A basic understanding of how the SIP and RTP protocols work together to create a VoIP call can be helpful in understanding and debugging functional problems (such as the "one-way audio" issue NAT configuration errors can often create). NAT allows a single external IP address to be shared by multiple devices behind a router. Since NAT is typically handled in the firewall, it also forms part of the security layer between a private network and the internet.

A VoIP call using SIP doesn't consist just of the signaling messages to set up the call (the SIP part of the connection). It also requires the RTP streams (the media), which carry the actual audio connection,[10] as shown in Figure 7-1.

6 We would again like to note that we've written extensively about digital circuits and DAHDI in previous editions, and that body of work has been released under Creative Commons licensing, and is freely available. Also, Sangoma/Digium provide detailed instructions on how to install and configure their PSTN cards. If you are looking to deploy this technology, please enlist the services of a professional technical resource. This stuff is complex and nuanced, and is not something you're going to enjoy playing with if you haven't had some sort of previous experience. It is not necessary to learning or understanding Asterisk.

7 When we say "lengthy" we mean that in relation to other electronic technologies.

8 Just try to imagine if your telephone number could be spammed the way your email address is. The fact that the PSTN is regulated, costs money to use, and controls the telephone numbers has served to limit the plague of spam that email has suffered.

9 The Wikipedia page on network address translation is comprehensive and useful. For more information about different types of NAT, and how NAT operates in general, start there.

10 SIP is not the only protocol to use RTP to carry media streams.

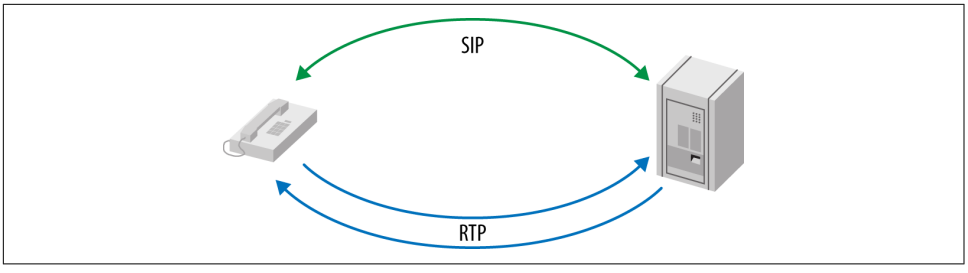

Figure 7-1. SIP and RTP

The use of separate protocols to carry the audio is what can make NAT traversal troublesome for VoIP connections, especially if the remote phones are behind one NAT, and the PBX is behind a different NAT. The problem is caused by the fact that while the SIP signaling will typically be allowed to pass through the firewalls at both ends, the RTP streams may not be recognized as part of the SIP session taking place, and thus will be ignored or blocked, as shown in Figure 7-2. The effect of one or both of the RTP streams being blocked is that users will complain that they are seeing their calls happen, and can answer them, but cannot hear (or cannot be heard).

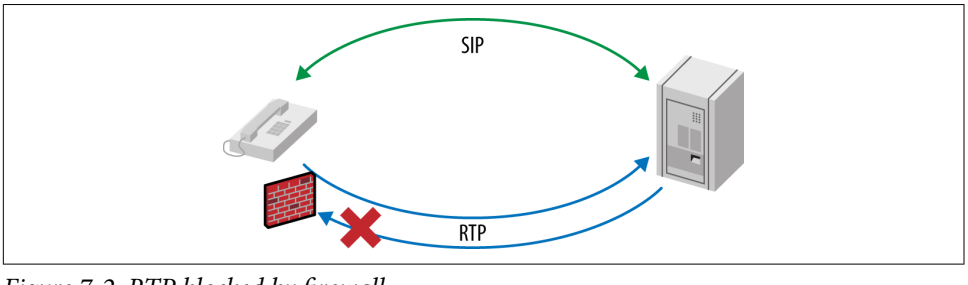

Figure 7-2. RTP blocked by firewall

In this section we will discuss some of the methods you may employ to alleviate issues caused by NAT. There are two different scenarios that need to be considered, each requiring you to define parameters within the *pjsip.conf* file. NAT issues can be annoying to troubleshoot, as there are many different types of firewalls in production, and many different ways to configure them.

In general, you're going to want to add the following to the transport sections of your */etc/asterisk/pjsip.conf* file:

```
[transport-udp]
type=transport
protocol=udp
bind=0.0.0.0
local_net=x.x.x.x/xx                ; IP/CIDR of your internal network
external_media_address=x.x.x.x      ; External IP address to use in RTP handling
external_signaling_address=x.x.x.x  ; External address for SIP signalling
```

If you want to find the external address of your PBX, run the following from the shell:

```
$ dig +short myip.opendns.com @resolver1.opendns.com
```

It is probably safe to set these parameters for all scenarios, but be prepared to experiment by commenting out settings and reloading PJSIP to test different scenarios.

Endpoints behind NAT

If your telephone sets are behind a remote NAT, there may be options in the `ps_endpoints` table of your database that should be adjusted from the default settings. You will need to experiment with changing the following values between `'yes'` and `'no'`. There may be many different combinations possible.

```
MySQL> update ps_endpoints set rtp_symmetric='yes',force_rport='yes',rewrite_contact='yes'
```

Other parameters you may wish to look at include `media_address` and `direct_media`.

Keep in mind the defaults when you are making changes. If in doubt, set the value of a field you've changed to `NULL`, as that will effectively set it back to the default.

Keeping a Remote Firewall Open

Sometimes a problem with a SIP telephone will surface wherein the phone will register and function when it is first booted, but then it will suddenly become unreachable. What is often happening here is that the remote firewall, seeing no activity coming from the set, will close the external connection to the telephone, and thus the PBX will lose the ability to signal to the set. The effect is that if the PBX tries to send a call to the phone, it will fail to connect (the remote firewall will reject the connection). If, on the other hand, the user makes a call out, for a few minutes the set will again be able to accept incoming calls. Naturally this can cause a lot of confusion for the users.

A relatively simple solution to this problem[11] involves setting the registration timer on the remote phone to a low enough value that it will stimulate the connection every minute or so, and thus convince the firewall that this connection can be allowed to exist for a little while longer. It's a bit of a hack, but it has proven successful. The challenge with proposing a universal solution is that there are many different models of firewalls, from inexpensive consumer-grade units to complex session border controllers, and this is one of the few solutions that seems to address the problem reliably in almost all cases.

11 Whether it is the best solution is still up for debate.

This approach is best on smaller systems (fewer than 100 telephones). A larger system with hundreds or thousands of phones will not be well served by this solution, as there will be an increased load on the system due to a near-constant flood of registrations from remote phones. In such a case, some more careful thought will need to be given to the overall design (for example, a dedicated registrar server could be used in place of Asterisk to handle the registration traffic).

In a perfect world, you would be able to specify a particular model of firewall, and devise a configuration for those firewalls that would ensure your SIP traffic was properly handled. In reality, you will come up against not just different models of firewall, but even different firmware versions for the same model firewall.

Asterisk behind NAT

First up, we need to tell you that putting your PBX behind a NAT is not recommended. It's far better to ensure it is firewalled without a NAT layer (especially if you have endpoints that are not on the same network as the PBX).

If you are stuck working with a PBX behind a NAT, you will need to work with your network team to ensure that the NAT components of the network are being correctly handled by your NAT device (typically your firewall). If they do not have sufficient skill in this regard, you may require the services of an outside consultant who is skilled in NAT traversal for SIP/RTP traffic. As we said, having your PBX behind NAT is not recommended.

Typically, the endpoints will also be behind NAT, and thus you will have a double-NAT scenario, which is likely to require a few hours of wrangling with various settings, not only in Asterisk but also in the firewall, in order to achieve success. Remember that it is vital that you test audio in both directions; it's not enough to simply verify that calls can be dialed and answered.

In a scenario where there is no choice but to use a double-NAT, we would recommend finding out whether a VPN can be used between the PBX and the remote endpoints. In many cases this will end up being easier to configure.

We wish we could say there's a simple, reliable way to ensure NAT works in all cases, but unfortunately there is not.

You could also look into using NAT assisting technologies such as STUN, TURN, and ICE. The details of these are beyond the scope of this book, since they require external servers, but many folks have had success with those protocols where other methods failed.

PSTN Termination and Origination

Passing calls between a VoIP environment and the PSTN requires some sort of gateway to convert the VoIP (typically SIP) signaling into something compatible with PSTN protocols. These processes are referred to as *origination* and *termination* (Figure 7-3).

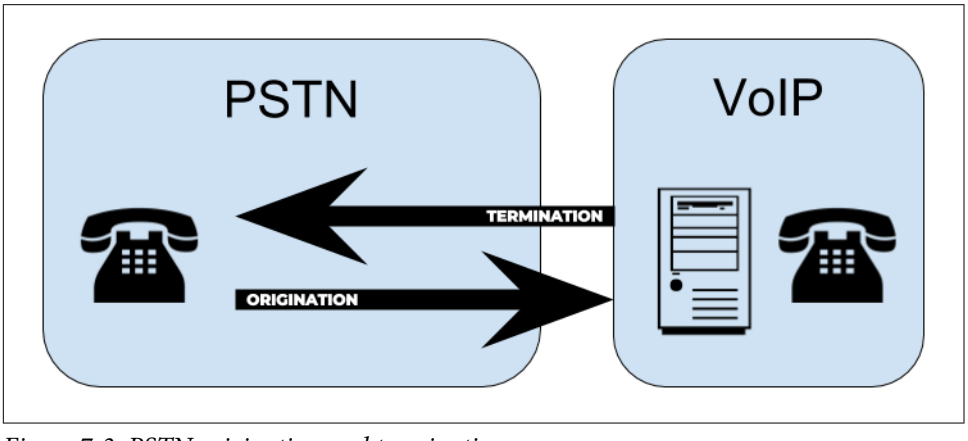

Figure 7-3. PSTN origination and termination

People often confuse the terms *origination* and *termination* as to which is which. For us, it's useful to remember that since the PSTN was already there when VoIP came along, the terms evolved in relation to it. Ideally, the processes should probably be called *PSTN origination*, and *PSTN termination*, and we encourage you to remember them that way.[12]

PSTN termination

Until VoIP totally replaces the PSTN, there will be a need to connect calls from VoIP networks to the public telephone network. This process is referred to as *termination* (Figure 7-4).

12 And perhaps even use them that way in conversation, since many people are confused by these terms, and few will admit it when you're talking to them.

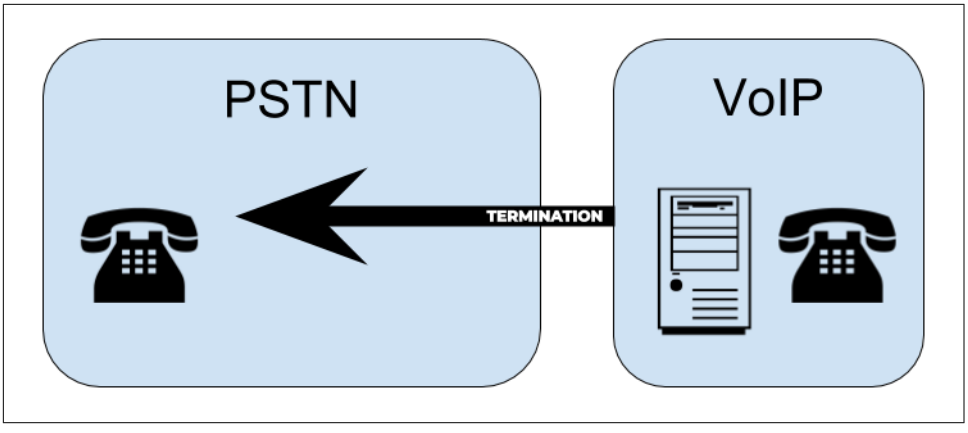

Figure 7-4. PSTN termination

Although you can engineer an Asterisk system to act as a termination gateway (using some sort of PSTN interfaces), in practice you're more likely to use an *Internet Telephony Service Provider* (ITSP, also sometimes called a *VoIP carrier*) to terminate your phone calls. ITSPs typically have a massive investment in infrastructure, and you'd be hard-pressed to do better without spending a ton of money. ITSPs have made termination inexpensive, so it's tough to make a business case for doing it yourself.

If you really need to connect your Asterisk system directly to the PSTN, you'll need the following:

- Appropriate circuit(s) from a PSTN telco (analog, BRI, PRI, SS7, MFC/R2, etc.)
- Suitable hardware to connect to that circuit (FXO, BRI, T1, E1, etc.)
- Echo cancellation (hardware or software)
- The skills necessary to properly configure your equipment for the carrier you're dealing with (there are many flavors of each of these circuit types, and this can be difficult to get right even for those who know the technology well)[13]

Beyond that, you will often have to handle a far more complex routing logic, which takes into consideration things like geography, corporate policy, cost, available resources, and so forth.

13 Trust us, Jim Van Meggelen worked with this stuff for many years before getting into VoIP.

In order to send your calls to an ITSP, your dialplan needs to look something like this:

```
; NANP-based systems
[to-pstn] ; Yes, we're going through an ITSP, but the PSTN is our destination
exten => _1NXXNXXXXXX.,1,Dial(${TOLL}/${EXTEN})  ; Country code plus phone number

; Add a '1' and send
exten => _NXXNXXXXXX.,1,Dial(${LOCAL}/1${EXTEN}) ; Country code plus phone number

; Strip off the '011' and send
exten => _011X.,1,Dial(${TOLL}/${EXTEN:3})       ; Country code plus phone number

; Emergency dialing
exten => 911,1,Dial(${LOCAL}/911) ; Defining this will require info from your carrier

; Most of the rest of the world
[to-pstn]
; Strip off NDD prefix, add country code, and send
exten => _0X.,1,Dial(${TOLL}/<add your country code here>${EXTEN:1})

; Strip off IDD prefix and send
exten => _00X,1,Dial(${LOCAL}/${EXTEN:2}) ; Country code plus phone number

; Emergency dialing (and other services)
exten => 11X,1,Dial(${LOCAL}/${EXTEN}) ; Defining this will require info from your carrier
```

Given that most PSTN circuits will allow you to dial any number, anywhere in the world, and given that you will be expected to pay for all incurred charges, we cannot stress enough the importance of security on any gateway machine that is providing PSTN termination. Criminals put a lot of effort into cracking phone systems (especially poorly secured Asterisk systems), and if you do not pay careful attention to all aspects of security, you will be the victim of toll fraud. It's only a matter of time.

Do not allow any unsecured VoIP connections into any context that contains PSTN termination.

Termination will tend to be more complex than we've outlined here—even if you're using an ITSP as your carrier—but the basic concept is fairly straightforward: match a pattern that your users might dial, prepare it for the carrier by removing or adding necessary digits, and send the call out the appropriate PJSIP endpoint (trunk). We've only discussed the dialplan here; in a later section we'll discuss how to configure the SIP trunks to carry this traffic.

PSTN origination

You might also want to be able to accept calls from the PSTN into your VoIP network. The process of doing this is commonly referred to as *origination*. This simply means that the call originated in the PSTN (Figure 7-5).

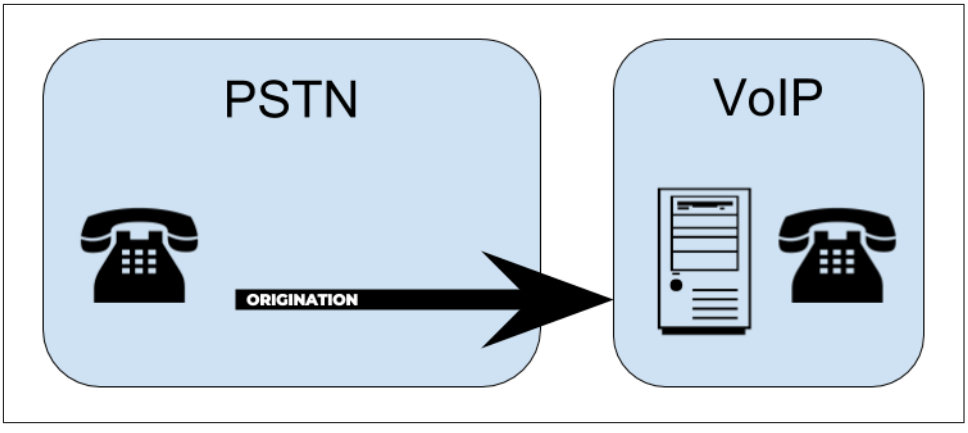

Figure 7-5. PSTN origination

In order to provide origination, a phone number is required.

In the good old days, when VoIP and Asterisk were new, it was quite common for people to handle the circuit connection to the PSTN themselves, using analog or digital trunks provided by the local phone company. For the most part this type of connection is now handled by ITSPs, and you simply need to connect your system to your VoIP carrier across a SIP trunk.

Phone numbers—when used for the purpose of origination—are commonly called DIDs (Direct Inward Dialing numbers). Your carrier will send a call down the circuit to your system, and pass the DID (or special received digits in some cases[14]), which the Asterisk dialplan will interpret. In other words, you will need a dialplan context that accepts incoming calls from your carrier, with extensions or patterns that will correlate to your DIDs.

In order to accept a call from a VoIP circuit, you will need to handle the digits the carrier will send you (the DID or phone number). The DNIS number and the DID do not have to match, but typically they will. In days gone by, the carrier would usually ask you in what format you wish to receive the digits. Nowadays, a VoIP carrier will

14 In traditional PBXs, the purpose of DIDs was to allow connection directly to an extension in the office. Many PBXs could not support concepts such as number translation or flexible digit lengths, and thus the carrier had to pass the extension number, rather than the number that was dialed (which was also referred to as the DNIS number, from Directory Number Information Service). For example, the phone number 416-555-1234 might have been mapped to extension 100, and thus the carrier would have sent the digits `100` to the PBX instead of the DNIS of `4165551234`. If you ever replace an old PBX with an Asterisk system, you may find this translation in place, and you'll need to obtain a list of mappings between the numbers that the caller dials and the numbers that are sent to the PBX. It was also common to see the carrier only pass the last four digits of the DNIS number, which the PBX then translates into an internal number. With VoIP trunks this will seldom be the case, but be aware that it is possible.

typically tell you what format they will send, and you are expected to accommodate. Two common formats are: DNIS (which is essentially the digits of the DID that was called) or E.164, which means that they'll be including the country code with the number.

In the dialplan, you associate the incoming circuit with a context that will know how to handle the incoming digits. As an example, it could look something like this:

```
[from-pstn]

exten => 4165550100,1,Goto(sets,100,1)
exten => 4165550101,1,Goto(sets,101,1)
exten => 4165550102,1,Goto(sets,102,1)
exten => 4165550103,1,Goto(sets,103,1)
exten => 4165554321,1,Goto(main-menu,${EXTEN},1)
exten => 4165559876,1,VoiceMailMain() ; a handy back door for listening
                                      ; to voice messages

exten => i,1,Verbose(2,Incoming call to invalid number)
```

In the `number-mapping` context, you explicitly list all of the DIDs that you expect to handle, plus an invalid handler for any DIDs that are not listed (you could send invalid numbers to reception, or to an automated attendant, or to some context that plays an invalid prompt).

Now we're ready to discuss how to configure trunks to carry your external traffic.

Configuring SIP Trunks

SIP is far and away the most popular of the VoIP protocols—so much so that the terms *VoIP* and *SIP* have almost come to mean the same thing. In previous editions of this book, we've looked at some of the other protocols that were popular at the time (primarily IAX2 and H.323), but for this edition there's no real reason anymore to discuss anything but SIP. The channel drivers for those older protocols are still available in Asterisk, but they're no longer supported.

The SIP protocol is peer-to-peer and does not really have a formal trunk specification. This means that whether you are connecting a single phone to your server or connecting two servers together, the SIP connections will be similar. Having said that, there are some differences in the style of how these resources can be configured, and there will definitely be a difference in how your dialplan handles routing across trunks.

Connecting an Asterisk system to a SIP provider

It is quite common to use the same ITSP carrier for termination and origination, but be aware that the two processes are unrelated to each other. If calls going in one direction pass your testing, that doesn't mean calls in the other direction are OK. If you change configuration, test routing both in and out, every time.

Many carriers will provide sample configurations for Asterisk. Unfortunately, these documents generally refer to the older `chan_sip` driver, which has been deprecated. Digium has designed a PJSIP wizard that is intended to greatly simplify carrier configuration. You can still configure ITSP trunks using the exact same methods we've shown before for configuring other endpoints (creating records in `ps_endpoint`, `ps_aors`, `ps_auths`, and so forth), but rather than hash over all that again, we are going to take a look at the wizard, since it consolidates several components into a single configuration file. We have found that since user endpoints change often, and carrier endpoints seldom do, it's often useful to configure carriers in a configuration file rather than in the database.

Before any config can be created, however, it's important to determine how the carrier will interact with your system. There are two fundamental models we have seen:

Password-based authentication, including registration[15]
: This is common in smaller carriers focused on the small-business market. This is also the type of service you would get if you were simply registering a SIP phone directly to a service.

IP-based authentication
: No password; no registration. This is more common with carriers that provide bulk trunking services to larger enterprises and resellers. (Typically these will also come with some sort of minimum commitment in terms of volume.) You will be expected to have solid SIP and networking skills.

These are not hard-and-fast rules, but they are the most common in our experience.

So there are two ways we might configure an ITSP in the */etc/asterisk/pjsip_wizard.conf* file.

First, if the carrier uses an IP address–based authentication, they will expect you to send your traffic from a static IP address (and should your address change, you will need to inform them so they can reconfigure their equipment). Your *pjsip_wizard.conf* file could then look something like this:

```
; ITSP uses IP address-based authentication
[itsp-no-auth]
type=wizard
remote_hosts=itsp.example.com
endpoint/context=pstn-in
endpoint/allow = !all,ulaw,g722
sends_registrations=no
accepts_registrations=no
```

15 Remember that registration is simply a mechanism whereby a SIP endpoint tells a registrar server where it is located. This is useful if your IP address changes, as might be the case on a consumer or small-business type of internet connection (such as DSL or cable).

```
sends_auth=no
accepts_auth=no
```

Alternatively, if your IP address changes frequently (or your carrier requires this method), you can have your system register to the carrier (which will require you to send authentication credentials to prove it's really you). Your calls will typically also be required to authenticate:

```
[itsp-with-auth]
type=wizard
remote_hosts=itsp.example.com
endpoint/context=pstn-in
endpoint/allow = !all,ulaw,g722
sends_registration=yes
accepts_registrations=no
sends_auth=yes
accepts_auth=no
outbound_auth/username=itsp_provided_username
outbound_auth/password= itsp_provided_password
```

Note that the names `[itsp-no-auth]` and `[itsp-with-auth]` have no built-in meaning to Asterisk. They become the PJSIP channel names to which you send your calls.

Configure trunks for termination. The PJSIP wizard has created the channel definitions we require for our carrier. To send a call, we only need to make a minor change to the `[globals]` section of our *extensions.conf* file, as follows:

```
[globals]
UserA_DeskPhone=PJSIP/0000f30A0A01
UserA_SoftPhone=PJSIP/SOFTPHONE_A
UserB_DeskPhone=PJSIP/0000f30B0B02
UserB_SoftPhone=PJSIP/SOFTPHONE_B

TOLL=PJSIP/itsp-no-auth
LOCAL=${TOLL}
;OR
;TOLL=PJSIP/itsp-with-auth
;LOCAL=${TOLL}
```

Configuring trunks for origination. For your incoming calls, you'll need a context in your */etc/asterisk/extensions.conf* file that matches the context specified for the ITSP channel. Let's assume we have two NANP DIDs, 4169671111 and 4167363636, and place the required code above the `[sets]` context:

```
TOLL=PJSIP/itsp-no-auth
LOCAL=${TOLL}
;OR
;TOLL=PJSIP/itsp-with-auth
;LOCAL=${TOLL}

[pstn-in]
exten => 4169671111,1,Dial(sets,100,1)
exten => 4167363636,1,Dial(sets,101,1)
```

```
[sets]
exten => 100,1,Dial(${UserA_DeskPhone})
```

In a small system, this is fairly easy to administer. In a larger system, you'd want to put the DIDs into a table in your database, and have the dialplan look up the required target. We'll be diving into databases a bit more later in the book.[16]

That's the gist of it as far as carrier interconnection is concerned. It can seem very complicated to set this up because there are a lot of options, but at a high level it's fairly straightforward. Problems are usually found to be minor configuration mismatches. Be methodical, and please, please, please be paranoid about security!

Emergency Dialing

In North America, people are used to being able to dial 911 in order to reach emergency services. Outside of North America, well-known emergency numbers are 112 and 999. If you make your Asterisk system available to people, you are obligated (in many cases regulated) to ensure that calls can be made to emergency services from any telephone connected to the system (even from phones that otherwise are restricted from making calls).

One of the essential pieces of information the emergency response organization needs to know is where the emergency is (e.g., where to send the fire trucks). In a traditional PSTN trunk, this information is already known by the carrier and is subsequently passed along to whatever local authority handles these tasks (in Canada and the US, these are called Public Safety Answering Points, or PSAP). With VoIP circuits things can get a bit more complicated, by virtue of the fact that they are not physically tied to any geographical location.

You need to ensure that your system will properly handle emergency calls from any phone connected to it, and you need to communicate what is available to your users. As an example, if you allow users to register to the system from softphones on their laptops, what happens if they are in a hotel room in another country, and somebody dials 911?[17]

The dialplan for handling emergency calls does not need to be complicated. In fact, it's far better to keep it simple. People are often tempted to implement all sorts of fancy functionality in the emergency services portions of their dialplans, but if a bug

16 A table to handle this would simply need a field for the DID, and three more for the target context, extension, and priority.

17 Don't assume this can't happen. When somebody calls 911, it's because they have an emergency, and it's not safe to assume that they're going to be in a rational state of mind. A recording that tells your softphone users what address the system is going to be sending to the PSAP may clue them in to the fact that the fire trucks aren't going to be sent to where they're needed. ("This telephone is registered to our Toronto system. Emergency services will be sent to our office at 301 Front St W. Press 1 to proceed with this call.").

in one of your fancy features causes an emergency call to fail, lives could be at risk. *This is no place for playing around*. The `[emergency-services]` section of your dialplan might look something like this:

```
[emergency-services]
exten => 911,1,Goto(dialpsap,1)
exten => 9911,1,Goto(dialpsap,1) ; some people will dial '9' because
                                 ; they're used to doing that from the PBX
exten => 999,1,Goto(dialpsap,1)
exten => 112,1,Goto(dialpsap,1)

exten => dialpsap,1,Verbose(1,Call initiated to PSAP!)
    same => n,Dial(${LOCAL}/911) ; REPLACE 911 HERE WITH WHATEVER
                                 ; IS APPROPRIATE TO YOUR AREA

[internal]
include => emergency-services   ; you have to have this in any context
                                ; that has users in it
```

In contexts where you know the users are not on-site (for example, remote users with their laptops), something like this might be best instead:

```
[no-emergency-services]
exten => 911,1,Goto(nopsap,1)
exten => 9911,1,Goto(nopsap,1) ; for people who dial '9' before external calls
exten => 999,1,Goto(nopsap,1)
exten => 112,1,Goto(nopsap,1)

exten => nopsap,1,Verbose(1,Call initiated to PSAP!)
    same => n,Playback(no-emerg-service) ; you'll need to record this prompt

[remote-users]
include => no-emergency-services
```

In North America, regulations have obligated many VoIP carriers to offer what is popularly known as *E911*.[18] When you sign up for their services, they will require address information for each DID that you wish to associate with outgoing calls. This address information will then be sent to the PSAP appropriate to that address, and your emergency calls should be handled the same way they would be if they were dialed on a traditional PSTN circuit.

The bottom line is that you need to make sure that the phone system you create handles emergency calls in accordance with local regulations and user expectations.

18 It's not actually the carrier that's offering this; rather it's a capability of the PSAP. E911 is also used on PSTN trunks, but since that happens without any involvement on your part (the PSTN carriers handle the paperwork for you), you are generally not aware that you have E911 on your local lines.

Conclusion

It is generally predicted that the PSTN will eventually disappear entirely. Before that happens, however, a distributed mechanism that is widely used and trusted will be needed to allow organizations and individuals to publish addressing information so they can be found. Any voice technology that does not use the PSTN is currently either a walled-garden proprietary product, or is the playground of spammers and criminals. We suspect the PSTN may be around for a while yet, and if so, then origination and termination will need to be part of your Asterisk system.

CHAPTER 8
Voicemail

Just leave a message, maybe I'll call.
—Joe Walsh

Before email and instant messaging became ubiquitous, voicemail was a popular method of electronic messaging. Even though most people now prefer text-based messaging systems, voicemail remains an essential component of any PBX.

Asterisk has a reasonably flexible voicemail system named Comedian Mail.[1] Voicemail in Asterisk is provided in the dialplan by the `app_voicemail.so` module.

A Caveat About Voicemail in Asterisk

The `app_voicemail` module is one of the oldest in Asterisk, and it suffers from many limitations, especially when compared to other modules that have enjoyed a steady evolution. The code itself is something few have the nerve to mess with, and thus new features for this module are unlikely to ever appear. You need to understand that `app_voicemail` doesn't just provide a dialplan application; there are all sorts of things that have to happen to ensure this all works, such as storage and file management, interaction with the email system of the operating system, time zone awareness, file formatting, security, plus a whole basket of various parameters that might need to be set. The `app_voicemail` module does all of that, and as a result it ends up being a sort of kludgy subsystem (on traditional PBXs, the voicemail was in fact a completely separate machine).

1 This name was a play on words, inspired in part by Nortel's voicemail system Meridian Mail. Nortel (and Meridian Mail) are gone, but Comedian Mail soldiers on.

Numerous attempts have been made to re-engineer voicemail, but they've all come up short. The reasons are simple: the level of work (and therefore cost) required to re-engineer it (in such a way as to satisfy the needs of a diverse community), coupled with a lack of cultural interest in voicemail technology in general, have (thus far) quickly killed any such initiative.

It's important to state that Asterisk voicemail works, and works well. You'll probably find it suitable to your needs. If you don't, the community will suggest that you are more than welcome to take a crack at re-engineering it.

Some of the features of Asterisk's voicemail system include:

- Unlimited password-protected voicemail boxes, each containing mailbox subfolders for organizing voicemail
- Different greetings for busy and unavailable states
- Default and custom greetings
- The ability to associate phones with more than one mailbox, and mailboxes with more than one phone
- Email notification of voicemail, with the voicemail optionally attached as an audio file
- Voicemail forwarding and broadcasts
- Message-waiting indicator (flashing light or stuttered dialtone) on many types of phones
- Company directory of employees, based on voicemail boxes

We're now going to take you on a tour of the essential parts of the voicemail configuration file, covering the settings in the general section, the various regional settings that are possible, integration of voicemail into your dialplan, and a brief under-the-hood look at how Asterisk stores voicemail in the Linux filesystem.

The voicemail.conf File

Since we've installed the table required for voicemail in the MySQL database, it is possible to create mailboxes there without any other configuration. It is, however, also possible to create mailboxes in an */etc/asterisk/voicemail.conf* file (this file also allows you to change various other default settings). We'll continue to use the database to create and manage users, since it is far more suited to the task, but we'll also explore the configuration file so that you can get a sense of the flexibility possible with Asterisk's voicemail.

The *voicemail.conf* file contains several sections where various default parameters can be adjusted. For the most part, you won't need to change any of these; however, you should take a look at the *~/src/asterisk-1.15.<your version>/configs/samples/voicemail.conf.sample* file. It contains useful information about various options that can be adjusted.

We have provided a simple *voicemail.conf* file next. If you wish to tweak the basic configuration beyond this, simply add or edit the relevant option.

An Initial voicemail.conf File

We recommend the following sample as a starting point. You can refer to *~/asterisk-complete/asterisk/11/configs/voicemail.conf.sample* for details on the various settings.

Place the following in a file named */etc/asterisk/voicemail.conf*:

```
; Voicemail Configuration

[general]
format=wav49|wav
serveremail=voicemail@shifteight.org
attach=yes
skipms=3000
maxsilence=10
silencethreshold=128
maxlogins=3
emaildateformat=%A, %B %d, %Y at %r
pagerdateformat=%A, %B %d, %Y at %r
sendvoicemail=yes ; Allow the user to compose and send a voicemail while inside

[zonemessages]
eastern=America/New_York|'vm-received' Q 'digits/at' IMp
central=America/Chicago|'vm-received' Q 'digits/at' IMp
central24=America/Chicago|'vm-received' q 'digits/at' H N 'hours'
military=Zulu|'vm-received' q 'digits/at' H N 'hours' 'phonetic/z_p'
european=Europe/Copenhagen|'vm-received' a d b 'digits/at' HM
```

Setting up a Linux server to handle the sending of email is a Linux administration task that is beyond the scope of this book. You will need to test your voicemail-to-email service to ensure that the email is being handled appropriately by the Mail Transfer Agent (MTA),[2] and that downstream spam filters are not rejecting the messages (one reason this might happen is if your Asterisk server is using a hostname in the email body that does not in fact resolve to it).

2 Also sometimes called a Message Transfer Agent.

You can create a massive and complex *voicemail.conf* file (and can even store user mailboxes in it), but we're going to focus on a few curated examples to keep things simple. You'll likely find that what we present will serve your needs quite well (and that the documentation samples will provide more detail, should you require it).

The [general] Section

The first section of the *voicemail.conf* file, `[general]`, allows you to define global settings. Many of these settings can be assigned on a per-mailbox setting. We've listed in Table 8-1 a few settings that we feel are the most important to consider.

Table 8-1. [general] section options for voicemail.conf

Option	Value/example	Notes
`format`	`wav49|gsm|wav`	For each format listed, Asterisk creates a separate recording in that format whenever a message is left. The benefit is that some transcoding steps may be saved if the stored format is the same as the codec used on the channel. We like WAV because it is the highest quality, and WAV49 because it is nicely compressed and easy to email. We don't like GSM due to its scratchy sound, but it enjoys some popularity.[a]
`serveremail`	`user@domain`	When an email is sent from Asterisk, this is the email address that it will appear to come from.[b]
`attach`	`yes,no`	If an email address is specified for a mailbox, this determines whether the message is attached to the email (if not, a simple message notification is sent, and the user will need to call into voicemail to retrieve their messages).
`maxmsg`	`9999`	By default, Asterisk only allows a maximum of 100 messages to be stored per user. For users who delete messages, this is no problem. For people who like to save their messages, this space can get eaten up quickly. With the size of hard drives these days, you could easily store thousands of messages for each user, so our current thinking is to set this to the maximum and let the users manage things from there. Be aware that old voicemail messages on a large system can waste a lot of hard drive space, after a few years of storing every message.
`maxsecs`	`600`	This type of setting was useful back when a large voicemail system might have only 40 MB[c] of storage: it was necessary to limit the system because it was easy to fill up the hard drive. This setting can be annoying to callers (although it does force them to get to the point, so some people like it). Nowadays, with terabyte drives common, there is no technical reason to limit the length of a message. Two considerations are: 1) if a channel gets hung in a mailbox, it's good to set some sort of value so it doesn't mindlessly record an endless, empty voice message, but 2) if a user wants to use her mailbox to record notes to herself, she won't appreciate it if you cut her off after 3 minutes. A setting somewhere between `600` seconds (10 minutes) and `3600` seconds (1 hour) will probably be about right.
`emailsubject`	`[PBX]: New message ${VM_MSGNUM} in mailbox ${VM_MAILBOX}`	When Asterisk sends an email, you can use this setting to define what the `Subject:` line of the email will look like. See the *voicemail.conf.sample* file for more details.

Option	Value/example	Notes
emailbody	Dear ${VM_NAME}:\n\n\tyou have a ${VM_DUR} long message (number ${VM_MSGNUM})\nin mailbox ${VM_MAILBOX} \n\n\t\t\t\t--Asterisk\n	When Asterisk sends an email, you can use this setting to define what the body of the email will look like. See the *voicemail.conf.sample* file for more details.
emaildateformat	%A, %d %B %Y at %H:%M:%S	This option allows you to specify the date format in emails. Uses the same rules as the C function STRFTIME.
pollmailboxes	no, yes	If the contents of mailboxes are changed by anything other than app_voicemail (such as external applications or another Asterisk system), setting this to yes will cause app_voicemail to poll all the mailboxes for changes, which will trigger proper message waiting indication (MWI) updates.
pollfreq	30	Used in concert with pollmailboxes, this option specifies the number of seconds to wait between mailbox polls.

[a] The separator that is used for each format option must be the pipe (|) character.

[b] Sending email from Asterisk can require some careful configuration, because many spam filters will find Asterisk messages suspicious and will simply ignore them. We talk more about how to set email for Asterisk in "Voicemail to Email" on page 144.

[c] Yes, you read that correctly: megabytes.

External Validation of Voicemail Passwords

By default, Asterisk does not validate user passwords to ensure they are at least somewhat secure. Anyone who maintains voicemail systems will tell you that a large percentage of mailbox users set their passwords to something like 1234 or 1111, or some other string that's easy to guess. Although fraud bots aren't typically interested in making mischief, having lousy passwords does represent a security hole in the voicemail system.

Since the app_voicemail.so module does not have the built-in ability to validate passwords, the settings externpass, externpassnotify, and externpasscheck allow you to validate them using an external program. Asterisk will call the program based on the path you specify, and pass it the following arguments:

```
mailbox context oldpass newpass
```

The script will then evaluate the arguments based on rules that you defined in the external script, and, accordingly, it should return to Asterisk a value of `VALID` for success or `INVALID` for failure (actually, the return value for a failed password can be anything except the words `VALID` or `FAILURE`). This value is typically printed to `stdout`. If the script returns `INVALID`, Asterisk will play an invalid-password prompt and the user will need to attempt something different.

Ideally, you would want to implement rules such as the following:

- Passwords must be a minimum of six digits in length
- Passwords must not be strings of repeated digits (e.g., `111111`)
- Passwords must not be strings of contiguous digits (e.g., `123456` or `987654`)

Asterisk comes with a simple script that will greatly improve the security of your voicemail system. It is located in the source code under the folder: */contrib/scripts/voicemailpwcheck.py*.

We strongly recommend that you copy it to your */usr/local/bin* folder (or wherever you prefer to put such things), and then uncomment the `externpasscheck=` option in your *voicemail.conf* file. Your voicemail system will then enforce the password security rules you have established.

Part of the `[general]` section is an area of supplementary options (referred to in the config file as *advanced options*, though there's not really anything advanced about them). These options (listed in Table 8-2) are defined in the same way as the other options in the `[general]` section, but what is significant about them is that they can also be defined on a per-mailbox basis, which would override whatever is defined under `[general]` for that particular setting. *In other words, the following options can be set in the database when you create a new mailbox.*

Table 8-2. A curated list of supplementary options for voicemail.conf

Option	Value/example	Notes
`tz`	`eastern`, `european`, etc.	Specifies the `zonemessages` name, as defined under `[zonemessages]` (discussed in the next section).
`locale`	`de_DE.utf8`, `es_US.utf8`, etc.	Used to define how Asterisk generates date/time strings in different locales. To determine the locales that are valid on your Linux system, type **`locale -a`** at the shell.
`attach`	`yes`, `no`	If an email address is specified for a mailbox, this determines whether the messages are attached to the email notifications (otherwise, a simple message notification is sent).

Option	Value/example	Notes
attachfmt	wav49, wav, etc.	If attach is enabled and messages are stored in different formats, this defines which format is sent with the email notifications. Often wav49 is a good choice, as it uses a better compression algorithm and thus will use less bandwidth, but doesn't sound crappy, as gsm does.
exitcontext	*context*	There are options that allow the callers to exit the voicemail system when they are in the process of leaving a message (for example, pressing 0 to get an operator). By default, the context the caller came from will be used as the exit context. If desired, this setting will define a different context for callers exiting the voicemail system.
review	yes, no	This should almost always be set to yes (even though it defaults to no). People get upset if your voicemail system does not allow them to review their messages prior to delivering them.
operator	yes, no	Best practice dictates that you should allow your callers to "zero out" from a mailbox, should they not wish to leave a message. Note that an o extension (not "zero," but "oh") is required in the exitcontext in order to handle these calls.
delete	no, yes	After an email message notification is sent (which could include the message itself), the message will be deleted. This option is risky, because the fact that a message was emailed is not a guarantee that it was received (spam filters seem to love to delete Asterisk voicemail messages). On a new system, leave this at no until you are certain that no messages are being lost due to spam filters.
nextaftercmd	yes, no	This handy little setting will save you some time, as it takes you directly to the next message once you've finished dealing with the current message.
passwordlocation	spooldir	If you want, you can have mailbox passwords stored in the spool folder for each mailbox.[a] One of the advantages of using the spooldir option is that it will allow you to define file #include statements in *voicemail.conf* (meaning you can store mailbox references in multiple files, as you can with, for example, dialplan code). This is not possible otherwise, because app_voicemail normally writes password changes to the filesystem, and cannot update a mailbox password stored outside of either *voicemail.conf* or the spool. If you do not use passwordlocation, you will not be able to define mailboxes outside of *voicemail.conf*, since password updates will not happen. Storing passwords in a file in the specific mailbox folder in the spool solves this problem.

[a] Typically the spool folder is */var/spool/asterisk*, and it can be defined in */etc/asterisk/asterisk.conf*.

The [zonemessages] Section

The next section of the *voicemail.conf* file is the [zonemessages] section. The purpose of this section is to allow time zone–specific handling of messages, so you can play back to the user messages with the correct timestamps. You can set the name of the zone to whatever you need. Following the zone name, you can define which time zone you want the name to refer to, as well as some options that define how timestamps are played back. You can look at the *~//src/asterisk-16.<TAB>/configs/samples/voicemail.conf.sample* file for syntax details. Asterisk includes the examples shown in Table 8-3. Any valid time zone known to the Linux system should be

configurable. Just use the Linux name for the zone, and then provide the details of how you want it handled.

Table 8-3. [zonemessages] section options for voicemail.conf

Zone name	Value/example	Notes
`eastern`	`America/New_York|'vm-received' Q 'digits/at' IMp`	This value would be suitable for the Eastern time zone (EST/EDT).
`central`	`America/Chicago|'vm-received' Q 'digits/at' IMp`	This value would be suitable for the Central time zone (CST/CDT).
`central24`	`America/Chicago|'vm-received' q 'digits/at' H N 'hours'`	This value would also be suitable for CST/CDT, but would play back the time in 24-hour format.
`military`	`Zulu|'vm-received' q 'digits/at' H N 'hours' 'phonetic/z_p'`	This value would be suitable for Universal Time Coordinated (Zulu time, formerly GMT).
`european`	`Europe/Copenhagen|'vm-received' a d b 'digits/at' HM`	This value would be suitable for Central European time (CEST).

Mailboxes

You can configure mailboxes in the *voicemail.conf* file, but it's not the recommended way. We're going to use the database to define your mailboxes.

The first thing we need to do is tell Asterisk that voicemail users are available in the database. We do that by editing the */etc/asterisk/extconfig.conf* file:

```
$ sudo vim /etc/asterisk/extconfig.conf

[settings] ; older mechanism for connecting all other modules to the database
ps_endpoints => odbc,asterisk
ps_auths => odbc,asterisk
ps_aors => odbc,asterisk
ps_domain_aliases => odbc,asterisk
ps_endpoint_id_ips => odbc,asterisk
ps_contacts => odbc,asterisk
voicemail => odbc,asterisk,voicemail
```

You should restart Asterisk to ensure this change has been applied (`$ sudo service asterisk restart`).

In the voicemail system, a mailbox must be defined with a context. This does not relate to any dialplan context; it's a label specific to voicemail that will determine what mailboxes will be grouped together, and is also used to name the folder in the spool that contains the various files associated with this mailbox (greeting, messages, envelopes, and so forth). Normally, you don't need to worry about this, as all mailboxes will end up in the `default` context. You really only need to define various contexts if you have a complex, multi-tenanted system, where there's a potential for extension overlap, or where you don't want certain groups of users exposed to other groups of users.

The `asterisk`.`voicemail` table offers many options; however, to create a mailbox there are only three fields that are required, plus two more that are recommended. The `context`, `mailbox`, and `password` fields are required, and `fullname` and `email` are strongly recommended. Here's a simple MySQL `INSERT` that'll create some mailboxes for you.

```
INSERT INTO `asterisk`.`voicemail` (context,mailbox,password,fullname,email)
VALUES
('default','100','486541','Russell Bryant', 'russell@shifteight.org'),
('default','101','957642','Leif Madsen', 'leif@shifteight.org'),
('default','102','656844','Jared Smith', 'jared@shifteight.org'),
('default','103','375416','Jim VanMeggelen', 'jim@shifteight.org')
;
```

The parts of the mailbox definition are:

`mailbox`
: This is the mailbox number. It is normal to ensure it corresponds with the extension number of the associated set.

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

  If the password is preceded by the hyphen (-) character, the user cannot change their mailbox password.

`fullname (FirstName LastName)`
: This is the name of the mailbox owner. The company directory uses the text in this field to allow callers to spell usernames. You only get one space, which is meant to delimit the first name from the last name, so if your last name is something like *Van Meggelen*, you'll put that in as `VanMeggelen`. Other punctuation characters might also cause problems. (We're looking at you, O'Reilly.)

`email address`
: This is the email address of the mailbox owner. Asterisk can send the voicemail to the specified email box.

The Asterisk directory cannot handle the concept of a surname that is anything other than a simple word. This means that family names such as *O'Reilly*, *Bryant-Madsen-Smith*, and yes, even *Van Meggelen* must have any punctuation characters and spaces removed before being added to *voicemail.conf*.

There are quite a few other options you can define for each user. It's unlikely you'll use many of them, but Table 8-4 contains a curated list of some that may be of use to you.

Table 8-4. Mailbox options

Option	Description
`delete`	After Asterisk sends the voicemail via email, the voicemail is deleted from the server. This option is useful for users who only want to receive voicemail via email. Valid options are `yes` or `no`. *Option can only be set per mailbox.*
`envelope`	Turns on or off envelope playback prior to playback of the voicemail message. Valid options are `yes` or `no`. Default is `yes`.
`exitcontext`	The dialplan context to exit to when pressing * or 0 from the `Voicemail()` application. Works in conjunction with the `operator` option as well. Must have an extension `a` in the context for exiting with *. Must have an extension `o` in the context for exiting with 0. You'll need to do a bit of design work before your dialplan will be able to handle this well, so it's best to leave it blank until you've had a chance to prototype everything you'll need to handle.
`forcegreeting`	Forces the recording of a greeting for new mailboxes. A new mailbox is determined by the mailbox number and password matching. Valid values are `yes` or `no`. Default is `no`.
`forcename`	Forces the recording of the person's name for new mailboxes. A new mailbox is determined by the mailbox number and password matching. Valid values are `yes` or `no`. Default is `no`.
`hidefromdir`	If set to `yes`, this mailbox will be hidden from the `Directory()` application. Default is `no`.
`locale`	Allows you to set the locale for the mailbox in order to control formatting of the date/time strings. See *voicemail.sample.conf* for more information.
`messagewrap`	Allows the first and last messages to wrap around (e.g., allow last message to wrap back to the first on the next message, or first message to wrap to the last message when going to the previous message). Valid options are `yes` or `no`. Default is `no`.
`minpassword`	Sets the minimum password length. Argument should be a whole number.
`nextaftercmd`	Skips to the next message after the user presses the 7 key (delete) or 9 key (save). Valid values are `yes` or `no`. Default is `yes`.
`operator`	Will allow the sender of a voicemail to hit 0 before, during, or after recording of a voicemail. Will exit to the `o` extension in the same context, or the context defined by the `exitcontext` option. Valid options are `yes` or `no`. Default is `no`. There are security risks associated with this, so it's best not to use it until you're certain the `exitcontext` does not allow calls to leave the system (i.e., end up making an expensive overseas call).
`passwordlocation`	By default, the password for voicemail is stored in the *voicemail.conf* file, and modified by Asterisk whenever the password changes. This may not be desirable, especially if you want to parse the password from an external location (or script). The alternate option for `passwordlocation` is `spooldir`, which will place the password for the voicemail user in a file called *secret.conf* in the user's voicemail spool directory. Valid options are `voicemail.conf` and `spooldir`. The default option is `voicemail.conf`.
`review`	When enabled, will allow the user recording a voicemail message to re-record their message. After pressing the # key to save their voicemail, they'll be prompted whether they wish to re-record or save the message. Valid options are `yes` or `no`. Default is `no`.

Option	Description
saycid	If enabled, and a prompt exists in */var/spool/asterisk/voicemail/recordings/callerids*, then that file will be played prior to the message, playing the file instead of saying the digits of the caller ID number. Valid options are yes or no. Default is no.
sayduration	Determines whether to play the duration of the message prior to message playback. Valid options are yes or no. Default is yes.
saydurationm	Allows you to set the minimum duration to play (in minutes). For example, if you set the value to 2, you will not be informed of the message length for messages less than 2 minutes long. Valid values are whole numbers. Default is 2.
searchcontexts	For applications such as Voicemail(), VoicemailMain(), and Directory(), the voicemail context is an optional argument. If the voicemail context is not specified, then the default is to only search the default context. With this option enabled, all contexts will be searched. This comes with a caveat that, if enabled, the mailbox number must be unique across all contexts—otherwise there will be a collision, and the system will not understand which mailbox to use. Valid options are yes and no. Default is no.
sendvoicemail	Allows the user to compose and send a voicemail message from within the VoicemailMain() application. Available as option 5 under the advanced menu. If this option is disabled, then option 5 in the advanced menu will not be prompted. Valid options are yes or no. Default is no.
tempgreetwarn	Enables a notice to the user when their temporary greeting is enabled. Valid options are yes or no. Default is no.
tz	Sets the time zone for a voicemail user (or globally). See */usr/share/timezone* for different available time zones. Not applicable if envelope=no.
volgain	The volgain option allows you to set volume gain for voicemail messages. The value is in decibels (dB). The *sox* application must be installed for this to work.

Voicemail Dialplan Integration

There are two primary dialplan applications provided by the app_voicemail.so module in Asterisk. The first, simply named VoiceMail(), does exactly what you would expect it to, which is to record a message in a mailbox. The second one, VoiceMailMain(), allows a user to log into a mailbox to retrieve messages.

The VoiceMail() Dialplan Application

When you want to pass a call to voicemail, you need to provide two arguments: the mailbox (or mailboxes) in which the message should be left, and any options relating to this, such as which greeting to play or whether to mark the message as urgent. The structure of the VoiceMail() command is this:

```
VoiceMail(mailbox[@context][&mailbox[@context][&...]][,options])
```

The options you can pass to VoiceMail() that provide a higher level of control are detailed in Table 8-5.

Table 8-5. VoiceMail() optional arguments

Argument	Purpose
b	Instructs Asterisk to play the busy greeting for the mailbox (if no busy greeting is found, the unavailable greeting will be played).
d([c])	Accepts digits to be processed by context c. If the context is not specified, it will default to the current context.
g(#)	Applies the specified amount of gain (in decibels) to the recording. Only works on DAHDI channels.
s	Suppresses playback of instructions to the callers after playing the greeting.
u	Instructs Asterisk to play the unavailable greeting for the mailbox (this is the default behavior).
U	Indicates that this message is to be marked as urgent. The most notable effect this has is when voicemail is stored on an IMAP server. In that case, the email will be marked as urgent. When the mailbox owner calls in to the Asterisk voicemail system, he should also be informed that the message is urgent.
P	Indicates that this message is to be marked as priority.

The `VoiceMail()` application sends the caller to the specified mailbox, so that they can leave a message. The mailbox should be specified as *`mailbox`*`@`*`context`*, where *`context`* is the name of the voicemail context (not the dialplan context). The option letters `b` or `u` can be added to request the type of greeting. If the letter `b` is used, the caller will hear the mailbox owner's *busy* message (if one exists). If the letter `u` is used, the caller will hear the mailbox owner's *unavailable* message (also assuming one exists). If no greeting exists, the system will generate a generic message: *The person at extension <mailbox> is unavailable. Please leave a message at the tone.*

In the dialplan we built in Chapter 6, we created several extensions. Consider this simple example extension 102, which allows people to call `UserB_DeskPhone`:

```
exten => 102,1,Dial(${UserB_DeskPhone},10)
  same => n,Playback(vm-nobodyavail)
  same => n,Hangup()
```

We faked a voicemail by playing a prompt that didn't actually do anything. Let's change that so the call goes to an actual mailbox instead. For now we'll just let voicemail play a generic greeting that the caller will hear. Remember, the second argument to the `Dial()` application is a timeout. If the call is not answered before the timeout expires, the call is sent to the next priority. We've got a 10-second timeout, and a new priority to send the caller to voicemail after the dial timeout:

```
exten => 102,1,Dial(${UserB_DeskPhone},10)
  same => n,Voicemail(${EXTEN}@default,u))
  same => n,Hangup()
```

We can do more if we wish, and change it so that if the user is busy (on another call), the caller will hear a busy message. To do this, we will make use of the `${DIALSTATUS}` variable, which contains one of several status values (type **`core show application Dial`** at the Asterisk console for a listing of all the possible values):

```
exten => 102,1,Dial(${UserA_SoftPhone})
  same => n,GotoIf($["${DIALSTATUS}" = "BUSY"]?busy:unavail)

  same => n(unavail),VoiceMail(101@default,u)
  same => n,Hangup()

  same => n(busy),VoiceMail(101@default,b)
  same => n,Hangup()
```

Now callers will get voicemail (with the appropriate greeting) if the user is either busy or unavailable. An alternative syntax is to use the `IF()` function to define which of the unavailable or busy messages to use:[3]

```
exten => 103,1,Dial(${UserB_SoftPhone})
   same => n,Voicemail(${EXTEN}@default,${IF($["${DIALSTATUS}" = "BUSY"]?b:u)})
   same => n,Hangup()
```

A slight problem remains, however, in that our users have no way of retrieving their messages, nor setting their greetings or any other voicemail options. We will remedy that in the next section.

The VoiceMailMain() Dialplan Application

Users can retrieve their voicemail messages, change their voicemail options, and record their voicemail greetings using the `VoiceMailMain()` application. `VoiceMailMain()` accepts two arguments: the mailbox number (and context if necessary), plus a few options. Both arguments are optional.

The structure of the `VoiceMailMain()` application looks like this:

```
VoiceMailMain([mailbox][@context][,options])
```

If you do not pass any arguments to `VoiceMailMain()`, it will play a prompt asking the caller to provide their mailbox number. The options that can be supplied are listed in Table 8-6.

Table 8-6. VoiceMailMain() optional arguments

Argument	Purpose
`p`	Allows you to treat the *mailbox* parameter as a prefix to the mailbox number.
`g(#)`	Increases the gain by # decibels when playing back messages.
`s`	Skips the password check.

3 We'll dive into functions like `IF()` in Chapter 10.

Argument	Purpose
a(*folder*)	Starts the session in one of the following voicemail folders (defaults to 0): • 0 - INBOX • 1 - Old • 2 - Work • 3 - Family • 4 - Friends • 5 - Cust1 • 6 - Cust2 • 7 - Cust3 • 8 - Cust4 • 9 - Cust5

To allow users to dial an extension to check their voicemail, you could add an extension to the dialplan like this:

```
exten => *98,1,NoOp(Access voicemail retrieval.)
   same => n,VoiceMailMain()
```

Any user whose device is assigned to the `[sets]` context can now dial *98, and they'll be able to log into their mailbox to listen to messages, record their name, set their greeting, and so forth.

Standard Voicemail Keymap

Figure 8-1 shows the standard keymap configuration for Asterisk Mail. Some options may be enabled or disabled based on the configuration of *voicemail.conf* (e.g., `envelope=no`). This can be given to users as a reference.

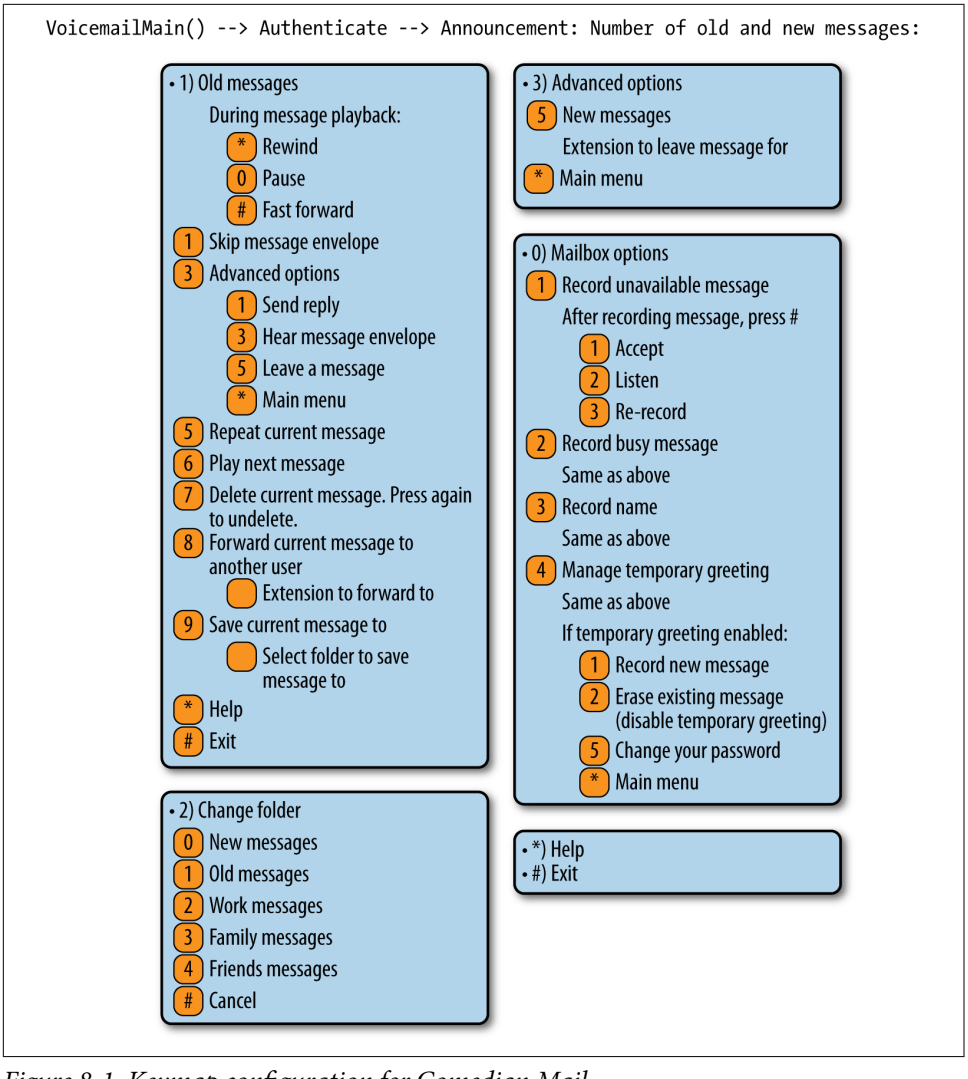

Figure 8-1. Keymap configuration for Comedian Mail

Creating a Dial-by-Name Directory

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

`Directory()` takes up to three arguments: the voicemail context from which to read the names, the optional dialplan context in which to dial the user, and an option

string (which is also optional). By default, `Directory()` searches for the user by last name, but passing the `f` option forces it to search by first name instead. Let's add two dial-by-name directories to the `TestMenu` context of our sample dialplan, so that callers can search by either first or last name:

```
exten => 4,1,Dial(${UserB_SoftPhone},10)
   same => n,Playback(vm-nobodyavail)
   same => n,Hangup()

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

exten => i,1,Playback(pbx-invalid)
   same => n,Goto(TestMenu,start,1)
```

If you call 201, and then press 8, you'll get a directory by first name. If you dial 9, you'll get the directory by last name.

Voicemail to Email

When Asterisk first came out, it did something very simple that was nevertheless revolutionary within the PBX market of the time. None of the major PBX brands could figure out how to effectively send voice messages to email (which, put simply, is just sending an email with the message itself as a WAV file attachment). Sure, some manufacturers offered the functionality, but it was needlessly complex, unreliable, and expensive. Asterisk cut through all that nonsense and just allowed a mailbox to have an assigned email address, and messages would simply be sent through the normal email mechanisms of Linux. This proved both simple and effective, and really showed how out-of-date and out-of-touch the traditional PBX manufacturers were.

Unfortunately, in every great story there's always a bad guy, and in this case a whole epidemic of them: spammers nearly brought the internet to its knees. The simple SMTP relay could no longer be trusted, as any machine open to relaying email would quickly become a vector for spam.

So, email became far more complex. If you want to send email from your Asterisk system, you have three fundamental ways to do that, as shown in Table 8-7.

Table 8-7. Overview of methods for transmitting voicemail to email

Method	Cons	Pros
1. Send email in the clear, directly to the SMTP port of the MX record the target domain returns. **Almost guaranteed to fail.**	Downstream spam filters will tend to discard suspicious traffic, and this traffic will look very suspicious.	No configuration required on the Asterisk server.
2. Relay your email through a host that knows and trusts your system. **Solid DNS and mail server skills are required by the team handling the relay server**.	The downstream relay server will need to be configured to work correctly with this arrangement (it will need to trust email relayed from your Asterisk server).	Relatively simple to configure on the Asterisk server.
3. Create a normal user account on an email server (complete with a valid email address), and send emails as an authenticated user through that platform. **We recommend this method** since it tends to work very well, and the requirements can be easily communicated to the team that maintains your email.	Slightly more complicated to set up on the Asterisk server.	Easy to set up an email account for Asterisk: you just have to create a user on your email system named "Company PBX" or some name that identifies it, and then use the credentials for this user to send all email through.

Essentially, what you need to do is make sure the Mail Transport Agent (MTA)[4] of your Asterisk server can send email from the `asterisk` shell/user account. The Asterisk voicemail engine will use the same mechanisms to send your voicemail to email.

For further information on the subject of MTAs, you'll want to consult a Linux administration book such as *UNIX and Linux System Administration Handbook*, 5th Edition, or an MTA-specific title such as O'Reilly's *Postfix: The Definitive Guide*.

Voicemail Storage Backends

The storage of messages on traditional voicemail systems has always tended to be overly complicated.[5] Asterisk not only provides you with a simple, logical filesystem-based storage mechanism, but also offers a few extra message storage options.

4 Popular MTAs these days are Postfix and Exim. The ubiquitous sendmail still exists as well, although its popularity has waned in the past few years. You'll find Postfix on your RHEL/CentOS machines by default, and likely Exim on your Debian/Ubuntu platforms (although Postfix is often recommended as the MTA there too).

5 Nortel used to store its messages in a sort of special partition, in a proprietary format, which made it pretty much impossible to extract messages from the system, or email them, or archive them, or really do anything with them. Ah, the good old days of closed, proprietary systems. We miss ... no ... wait ... we do not miss them!

Linux Filesystem

By default, Asterisk stores voice messages in the spool, at */var/spool/asterisk/voicemail/<voicemailcontext>/<mailbox>*. The messages can be stored in multiple formats (such as `wav` and `wav49`), depending on what you specified as the `format` in the `[general]` section of your *voicemail.conf* file. Your greetings are also stored in this folder.

Asterisk does not create a folder for any mailboxes that do not have any recordings yet (as would be the case with a new mailbox), so this folder cannot be used as a reliable method of determining which mailboxes exist on the system.

Figure 8-2 shows an example of what might be in a mailbox folder. This mailbox has no new messages in the *INBOX*, has two saved messages in the *Old* folder, and has *busy*, *unavailable* and name (*greet*) greetings recorded.

```
/var/spool/asterisk/voicemail/default/301
     ./INBOX
     ./Old
          ./Old/msg0000.WAV
          ./Old/msg0000.txt
          ./Old/msg0001.WAV
          ./Old/msg0001.txt
     ./Urgent
     ./busy.WAV
     ./unavail.WAV
     ./greet.WAV
```

Figure 8-2. Sample mailbox folder

For each message, there is a matching *msg####.txt* file, which contains the envelope information for the message. The *msg####.txt* file is also critically important for message waiting indication (MWI), as this is the file that Asterisk looks for in the *INBOX* to determine whether the message light for a user should be on or off.

IMAP

Some organizations prefer to manage voicemail as part of their email system. This has been called *unified messaging* by the telecom industry, and its implementation has traditionally been expensive and complex. Asterisk allows for a fairly simple integration between voicemail and email, either through its built-in voicemail-to-email handler, or through a relationship with an IMAP server. We don't recommend IMAP integration simply because it's a lot of work for very little gain, and it is out of scope for this book.

Message Storage in a Database

It is possible to configure Asterisk voicemail to store messages as blobs within a database. This was originally seen as a simple way to allow synchronization of messages between systems. We've never been fans of the idea, since databases are not designed for bulk storage of binary data, and there are many other ways to synchronize files across systems.

Conclusion

Asterisk's voicemail system is a mature and capable module, and an essential part of any PBX. It's not likely to be enhanced beyond what it does, but that's not likely to be a problem, either.

CHAPTER 9
Internationalization

David Duffett

English? Who needs to spend time learning that? I'm never going to England!
—Dan Castellaneta

Telephony is one of those areas of life where, whether at home or at work, people do not like surprises. When people use phones, anything outside of the norm is an expectation not met, and as someone who is probably in the business of supplying telephone systems, you will know that expectations going unmet can lead to untold misery in terms of the extra work, lost money, and other problems that are associated with customer dissatisfaction.

In addition to ensuring that the user experience is in keeping with what users expect, there is also the need to make your Asterisk feel "at home." For example, if an outbound call is placed over an analog line (FXO), Asterisk will need to interpret the tones that it "hears" on the line (busy, ringing, etc.).

By default (and maybe as one might expect, since it was "born in the USA"), Asterisk is configured to work within North America. However, since Asterisk gets deployed in many places and (thankfully) people from all over the world make contributions to it, it is quite possible to tune Asterisk for correct operation just about anywhere you choose to deploy it.

If you have been reading this book from the beginning, chapter by chapter, you will have already made some choices during installation and in the initial configuration that will have set up your Asterisk to work in your local area (and live up to your customers' expectations).

Quite a few of the chapters in this book contain information that will help you internationalize[1] or (perhaps more properly) localize your Asterisk implementation. The purpose of this chapter is to provide a single place where all aspects of the changes that need to be made to your Asterisk-based telephone system in this context can be referenced, discussed, and explained. The reason for using the phrase "Asterisk-based telephone system" rather than just "Asterisk" is that some of the changes will need to be made in other parts of the system (IP phones, ATAs, etc.), while other changes will be implemented within Asterisk and DAHDI configuration files.

Let's start by getting a list together (in no particular order) of the things that may need to be changed in order to optimize your Asterisk-based telephone system for a given location outside of North America. You can shout some out if you like…

- Language/accent of the prompts
- Physical connectorization for PSTN interfaces (FXO, BRI, PRI)
- Tones heard by users of IP phones and/or ATAs
- Caller ID format sent and/or received by analog interfaces
- Tones for analog interfaces to be supplied or detected by Asterisk
- Format of time/date stamps for voicemail
- The way the above time/date stamps are announced by Asterisk
- Patterns within the dialplan (of IP phones, ATAs, and Asterisk itself if you are using the sample dialplan)
- The way to indicate to an analog device that voicemail is waiting (MWI)
- Tones supplied to callers by Asterisk (these come into play once a user is "inside" the system; e.g., the tones heard during a call transfer)

We'll cover everything in this list, adopting a strategy of working from the outer edge of the system toward the very core (Asterisk itself). We will conclude with a handy checklist of what you may need to change and where to change it.

Although the principles discussed in this chapter will allow you to adapt your Asterisk installation specifically for your region (or that of your customer), for the sake of consistency, all of our examples will focus on how to adapt Asterisk for one region: the United Kingdom.

1 *i18n* is a term used to abbreviate the word *internationalization*, due to its length. The format is <*first_letter*><*number*><*last_letter*>, where <*number*> is the number of letters between the first and last letters. Other words, such as localization (L10n) and modularization (m12n), have also found a home with this scheme, which Leif finds a little bit ridiculous. More information can be found in the W3C glossary online.

Devices External to the Asterisk Server

There are massive differences between a good old-fashioned analog telephone and any one of the large number of IP phones out there, and we need to pick up on one of the really fundamental differences in order to throw light on the next explanation, which covers the settings we might need to change on devices external to Asterisk, such as IP phones.

Have you ever considered the fact that an analog phone is a totally dumb device (we know that a basic model is very, very cheap) that needs to connect to an intelligent network (the PSTN), whereas an IP phone (e.g., SIP or IAX2) is a very intelligent device that connects to a dumb network (the internet or any regular IP network)? Figures 9-1 and 9-2 illustrate the difference.

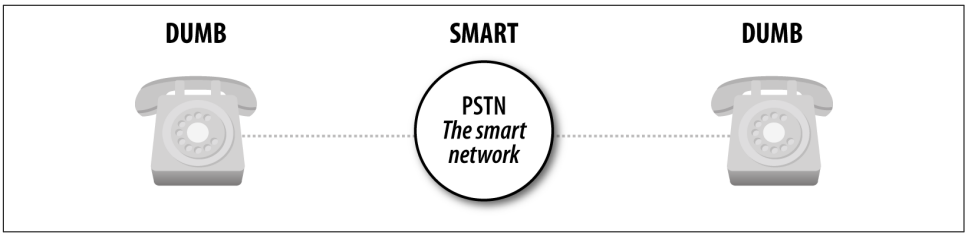

Figure 9-1. The old days: dumb devices connect to a smart network

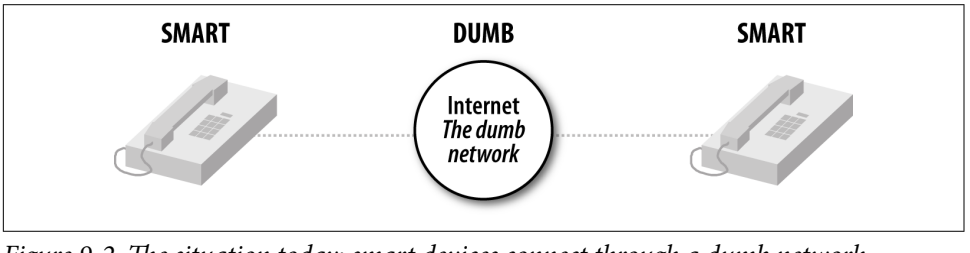

Figure 9-2. The situation today: smart devices connect through a dumb network

Could we take two analog phones, connect them directly to each other, and have the functionality we would normally associate with a regular phone? No, of course not, because the network supplies everything: the actual power to the phone, the dialtone (from the local exchange or CO), the caller ID information, the ringing tone (from the remote [closest to the destination phone] exchange or CO), all the signaling required, and so on.

Conversely, could we take two IP phones, connect them directly to each other, and get some sensible functionality? Sure we could, because all the intelligence is inside the IP phones themselves—they provide the tones we hear (dialtone, ringing, busy) and run the protocol that does all the required signaling (usually SIP). In fact, you can try this for yourself—most midpriced IP phones have a built-in Ethernet switch,

so you can actually connect the two IP phones directly to each other with a regular (straight-through) Ethernet patch cable, or just connect them through a regular switch. They will need to have fixed IP addresses in the absence of a DHCP server, and you can usually dial the IP address of the other phone just by using the * key for the dots in the address.

Figure 9-2 points to the fact that on an IP phone, we are responsible for setting all of the tones that the network would have provided in the old days. This can be done in one of (at least) two ways. The first is to configure the tones provided by the IP phone on the device's own web GUI. You do this by browsing to the IP address of the phone (the IP address can usually be obtained by a menu option on the phone) and then selecting the appropriate options. For example, on a Yealink IP phone, the tones are set on the *Phone* page of the web GUI, under the *Tones* tab (where you'll find a list of the different types of tones that can be changed—in the case of the Yealink, these are Dial, Ring Back, Busy, Congestion, Call Waiting, Dial Recall, Record, Info, Stutter, Message, and Auto Answer).

The other way that this configuration can be applied is to autoprovision the phone with these settings. A full explanation of the mechanism for autoprovisioning is beyond the scope of this book, but you can usually set up the tones in the appropriate attributes of the relevant elements in the XML file.

While we are changing settings on the IP phones, there are two other things that may need to be changed in order for the phones to look right and to function correctly as part of the system.

Most phones display the time when idle and, since many people find it particularly annoying when their phones show the wrong time, we need to ensure that the correct local time is displayed. It should be fairly easy to find the appropriate page of the web GUI (or XML attributes) to specify the time server. You will also find that there are settings for daylight saving time and other relevant stuff nearby.

The last thing to change is a potential showstopper as far as the making of a phone call is concerned—the dialplan. We're not talking about the dialplan we find in */etc/asterisk/extensions.conf*, but the dialplan of the phone. Not everyone realizes that IP phones have dialplans, too—although these dialplans are more concerned with which dial strings are permitted than with what to do on a given dial.

The general rule seems to be that if you dial on-hook the built-in dialplan is bypassed, but if you pick up the handset the phone's dialplan comes into play, and it just might happen that the dialplan will not allow the dial string you need to be dialed. Although this problem can manifest itself with a refusal by the phone to pass certain types of numbers through to Asterisk, it can also affect any feature codes you plan to use. This can easily be remedied by Googling the model number of the phone along with "UK dialplan" (or the particular region you need), or you can go to the appropriate page

on the web GUI and either manually adjust the dialplan or pick the country you need from a drop-down box (depending on the type of phone you are working with).

The prior discussion of IP phone configuration also applies to any analog telephone adapters (ATAs) you plan to use—specifically, to those supporting an FXS interface. In addition, you may need to specify some of the electrical characteristics of the telephony interface, like line voltage and impedance, together with the caller ID format that will work with local phones. All that differs is the way you obtain the IP address for the web GUI—you usually do this by dialing a specific code on the attached analog phone, which results in the IP address being read back to the caller.

Of course, an ATA may also feature an FXO interface, which will also need to be configured to properly interact with the analog line provided in your region. The types of things that need to be changed are similar to the FXS interface.

What if you are connecting your analog phone or line to a Digium card? We'll cover this next.

PSTN Connectivity, DAHDI, Digium Cards, and Analog Phones

Before we get to DAHDI and Asterisk configuration, we need to physically connect to the PSTN. Unfortunately, there are no worldwide standards for these connections; in fact, there are often variations from one part of a given country to another.

Primary Rate Interfaces (PRIs) are generally terminated in an RJ45 connection these days, although the impedance of the connections can vary. In some countries (notably in South America), it is still possible to find PRIs terminated in two BNC connectors, one for transmit and one for receive.

Generally speaking, a PRI terminated in an RJ45 will be an ISDN connection, and if you find the connection is made by a pair of BNC connectors (push-and-twist coaxial connectors), the likelihood is that you are dealing with an older CAS-based protocol (like MFCR2).

Figure 9-3 shows the adapter required if your telco has supplied BNC connectors (Sangoma/Digium cards require an RJ45 connection). It is called a *balun*, as it converts from a balanced connection (RJ45) to an unbalanced connection (the BNCs), in addition to changing the connection impedance.

Basic Rate Interfaces (BRIs) are common in continental Europe and are almost always supplied via an RJ45 connection.

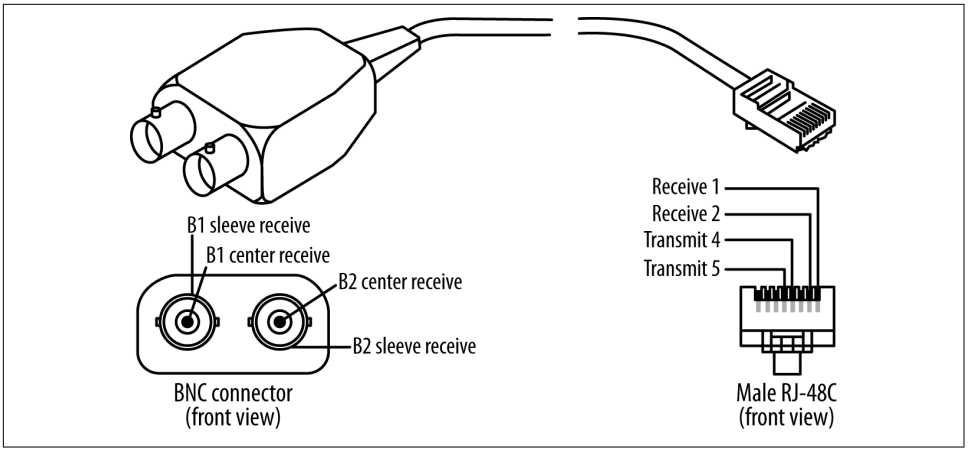

Figure 9-3. A balun

Analog connections vary massively from place to place—you will know what kind of connector is used in your locality. The important thing to remember is that the analog line is only two wires, and these need to connect to the middle two pins of the RJ11 plug that goes into the Digium card—the other end is the local one. Figure 9-4 shows the plug used in the UK, where the two wires are connected to pins 2 and 5.

Figure 9-4. The BT plug used for analog PSTN connections in the UK (note only pins 2–5 are present)

The Digium Asterisk Hardware Device Interface, or DAHDI, actually covers a number of things. It contains the kernel drivers for telephony adapter cards that work within the DAHDI framework, as well as automatic configuration utilities and test tools. These parts are contained in two separate packages (*dahdi-linux* and *dahdi-tools*), but we can also use one complete package, called *dahdi-linux-complete*. All three packages are available at the Digium site.

Once you have established the type of PRI connection the telco has given you, there are some further details that you will require in order to properly configure DAHDI and Asterisk (e.g., whether the connection is ISDN or a CAS-based protocol). Again, you will find these in Chapter 7.

DAHDI Drivers

The connections where some real localization will need to take place are those of analog interfaces. For the purposes of configuring your Asterisk-based telephone system to work best in a given locality, you will first need to specifically configure some low-level aspects of the way the Digium card interacts with the connected device or line. This is done through the DAHDI kernel driver(s), in a file called */etc/dahdi/system.conf*.

In the following lines (taken from the sample configuration that you get with a fresh install of DAHDI), you will find both the `loadzone` and `defaultzone` settings. The `loadzone` setting allows you to choose which tone set(s) the card will both generate (to feed to analog telephones) and recognize (on the connected analog telephone lines):

```
# Tone Zone Data
# ^^^^^^^^^^^^^^
# Finally, you can preload some tone zones, to prevent them from getting
# overwritten by other users (if you allow non-root users to open /dev/dahdi/*
# interfaces anyway). Also this means they won't have to be loaded at runtime.
# The format is "loadzone=<zone>" where the zone is a two-letter country code.
#
# You may also specify a default zone with "defaultzone=<zone>" where zone
# is a two-letter country code.
#
# An up-to-date list of the zones can be found in the file zonedata.c
#
loadzone = us
#loadzone = us-old
#loadzone=gr
#loadzone=it
#loadzone=fr
#loadzone=de
#loadzone=uk
#loadzone=fi
#loadzone=jp
#loadzone=sp
#loadzone=no
#loadzone=hu
#loadzone=lt
#loadzone=pl
defaultzone=us
#
```

The */etc/dahdi/system.conf* file uses the hash symbol (`#`) to indicate a comment instead of a semicolon (`;`) like the files in */etc/asterisk*.

Although it is possible to load a number of different tone sets (you can see all the sets of tones in detail in *zonedata.c*) and to switch between them, in most practical situations you will only need:

```
loadzone=uk     # to load the tone set
defaultzone=uk  # to default DAHDI to using that set
```

…or whichever tones you need for your region.

If you perform a *dahdi_genconf* to automatically (or should that be auto-magically?) configure your DAHDI adapters, you will notice that the newly generated */etc/dahdi/system.conf* will have defaulted both `loadzone` and `defaultzone` to being `us`. Despite the warnings not to hand-edit the file, it is fine to change these settings to what you need.

In case you were wondering how we tell whether there are any voicemails in the mailbox associated with the channel an analog phone is plugged into, it is done with a stuttered dialtone. The format of this stuttered dialtone is decided by the `loadzone`/`defaultzone` combination you have used.

As a quick aside, analog phones that have a message-waiting indicator (e.g., an LED or lamp that flashes to indicate new voicemail) achieve this by automatically going off-hook periodically and *listening* for the stuttered dialtone. You can witness this by watching the Asterisk command line to see the DAHDI channel go active (if you have nothing better to do!).

That's it at the DAHDI level. We chose the protocol(s) for PRI or BRI connections, the type of signaling for the analog channels (all covered in Chapter 7), and the tones for the analog connections that have just been discussed.

The relationship between Linux, DAHDI, and Asterisk (and therefore */etc/dahdi/system.conf* and */etc/asterisk/chan_dahdi.conf*) is shown in Figure 9-5.

Once you have completed your configuration at the DAHDI level (in */etc/dahdi/system.conf*), you need to perform a *dahdi_cfg -vvv* to have DAHDI reread the configuration. This is also a good time to use *dahdi_tool* to check that everything appears to be in order at the Linux level.

This way, if things do not work properly after you have configured Asterisk to work with the DAHDI adapters, you can be sure that the problem is confined to *chan_dahdi.conf* (or an *#included dahdi-channels.conf* if you are using this part of the *dahdi_genconf* output).

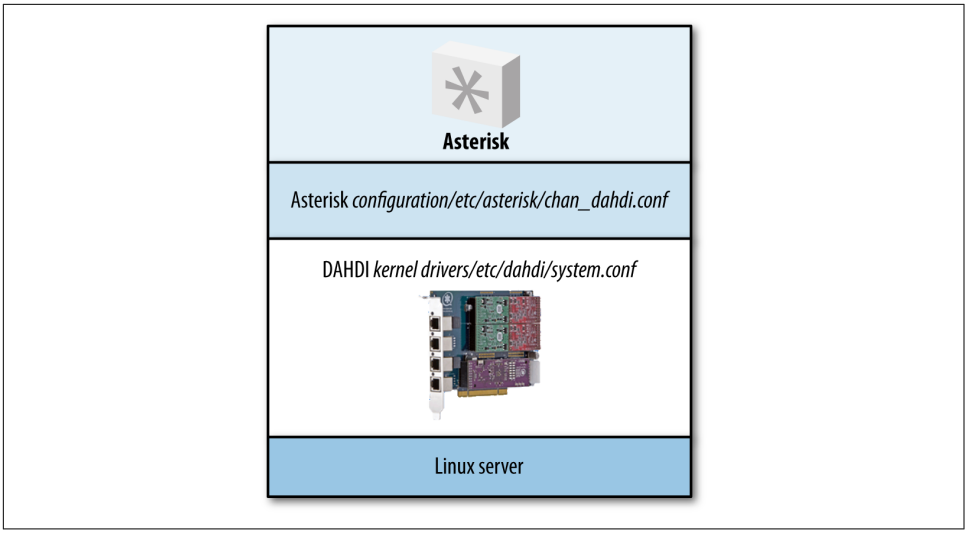

Figure 9-5. The relationship between Linux, DAHDI, and Asterisk

Internationalization Within Asterisk

With everything set at the Linux level, we now only need to configure Asterisk to make use of the channels we just enabled at the Linux level and to customize the way that Asterisk interprets and generates information that comes in from, or goes out over, these channels. This work is done in */etc/asterisk/chan_dahdi.conf*.

In this file we will not only tell Asterisk what sort of channels we have (these settings will fit with what we already did in DAHDI), but also configure a number of things that will ensure Asterisk is well suited to its new home.

Caller ID

A key component of this change is caller ID. While caller ID delivery methods are pretty much standard within the BRI and PRI world, they vary widely in the analog world; thus, if you plugged an American analog phone into the UK telephone network, it would actually work as a phone, but caller ID information would not be displayed. This is because that information is transmitted in different ways in different places around the world, and an American phone would be looking for caller ID signaling in the US format, while the UK telephone network would be supplying it in the UK format (if it is enabled—caller ID is not standard in the UK; you have to ask for and sometimes even pay for, it!).

Not only is the format different, but the method of telling a telephone (or Asterisk) to look out for the caller ID may vary from place to place, too. This is important, as we

do not want Asterisk to waste time looking for caller ID information if it is not being presented on the line.

Again, Asterisk defaults to the North American caller ID format (no entries in */etc/asterisk/chan_dahdi.conf* describe this, it's just the default), and in order to change it we will need to make some entries that describe the technical details of the caller ID system. In the case of the UK, the delivery of caller ID information is signaled by a polarity reversal on the telephone line (in other words, the A and B legs of the pair of telephone wires are temporarily switched over), and the actual caller ID information is delivered in a format known as V.23 (*frequency shift keying*, or FSK). So the entries in *chan_dahdi.conf* to receive UK-style caller ID on any FXO interfaces will look like this:

```
cidstart=polarity   ; the delivery of caller ID will be
                    ; signaled by a polarity reversal
cidsignalling=v23   ; the delivery of the called ID information
                    ; will be in V23 format
```

Of course, you may also need to send caller ID using the same local signaling information to any analog phones that are connected to FXS interfaces, and one more entry may be needed, as in some locations the caller ID information is sent after a specified number of rings. If this is the case, you can use this entry:

```
sendcalleridafter=2
```

Before you can make these entries, you will need to establish the details of your local caller ID system (someone from your local telco or Google could be your friend here, but there is also some good information in the sample */etc/asterisk/chan_dahdi.conf* file).

Language and/or Accent of Prompts

As you may know, the prompts (or recordings) that Asterisk will use are stored in */var/lib/asterisk/sounds*. In older versions of Asterisk all the sounds were in this actual directory, but these days you will find a number of subdirectories that allow the use of different languages or accents. The names of these subdirectories are arbitrary; you can call them whatever you want.

Note that the filenames in these directories must be what Asterisk is expecting—for example, in */var/lib/asterisk/sound/en*, the file *hello.gsm* would contain the word "Hello" (spoken by the lovely Allison), whereas *hello.gsm* in */var/lib/asterisk/sounds/es* (for Spanish in this case) would contain the word "Hola" (spoken by the Spanish equivalent of the lovely Allison[2]).

2 Who is, in fact, the same Allison who does the English prompts; June Wallack does the French prompts. The male Australian-accented prompts are done by Cameron Twomey. All voiceover talent are available to record additional prompts as well. See the Digium IVR page for more information.

The default directory used is */var/lib/asterisk/sounds/en*, so how do you change that?

There are two ways. One is to set the language in the channel configuration file that calls are arriving through using the `language` directive. For example, the line:

```
language=en_UK
```

placed in *chan_dahdi.conf*, *sip.conf*, and so on (to apply generally, or for just a given channel or profile) will tell Asterisk to use sound files found in */var/lib/asterisk/sounds/en_UK* (which could contain British-accented prompts) for all calls that come in through those channels.

The other way is to change the language during a phone call through the dialplan. This (along with many attributes of an individual call) can be set using the `CHANNEL()` dialplan function. See Chapter 10 for a full treatment of dialplan functions.

The following example would allow the caller to choose one of three languages in which to continue the call:

```
; gives the choice of (1) French, (2) Spanish, or (3) German
exten => s,1,Background(choose-language)
   same => n,WaitExten(5)

exten => 1,1,Set(CHANNEL(language)=fr)

exten => 2,1,Set(CHANNEL(language)=es)

exten => 3,1,Set(CHANNEL(language)=de)

; the next priority for extensions 1, 2, or 3 would be handled here
exten => _[123],n,Goto(menu,s,1)
```

If the caller pressed 1, sounds would be played from */var/lib/asterisk/sounds/fr*; if he pressed 2, the sounds would come from */var/lib/asterisk/sounds/es*; and so on.

As already mentioned, the names of these directories are arbitrary and do not need to be only two characters long—the main thing is that you match the name of the subdirectory you have created in the `language` directive in the channel configuration, or when you set the `CHANNEL(language)` argument in the dialplan.

Time/Date Stamps and Pronunciation

Asterisk uses the Linux system time from the host server, as you would expect, but we may have users of the system who are in different time zones, or even in different countries. Voicemail is where the rubber hits the road, as this is where users come into contact with time/date stamp information.

Consider a scenario where some users of the system are based in the US, while others are in the UK.

As well as the time difference, another thing to consider is the way people in the two locations are used to hearing date and time information—in the US, dates are usually ordered month, day, year, and times are specified in 12-hour clock format (e.g., 2:54 P.M.).

In contrast, in the UK dates are ordered day, month, year, and times are often specified in 24-hour clock format (14:54 hrs)—although some people in the UK prefer 12-hour clock format, so we will cover that, too.

Since all these things are connected to voicemail, you would be right to guess that we configure it in */etc/asterisk/voicemail.conf*—specifically, in the `[zonemessages]` section of the file.

Here is the `[zonemessages]` part of the sample *voicemail.conf* file that comes with Asterisk, with `UK24` (for UK people that like 24-hour clock format times) and `UK12` (for UK people that prefer 12-hour clock format) zones added:

```
[zonemessages]
; Users may be located in different time zones, or may have different
; message announcements for their introductory message when they enter
; the voicemail system. Set the message and the time zone each user
; hears here. Set the user into one of these zones with the tz=attribute
; in the options field of the mailbox. Of course, language substitution
; still applies here so you may have several directory trees that have
; alternate language choices.
;
; Look in /usr/share/zoneinfo/ for names of timezones.
; Look at the manual page for strftime for a quick tutorial on how the
; variable substitution is done on the values below.
;
; Supported values:
; 'filename' filename of a soundfile (single ticks around the filename
; required)
; ${VAR} variable substitution
; A or a Day of week (Saturday, Sunday, ...)
; B or b or h Month name (January, February, ...)
; d or e numeric day of month (first, second, ... thirty-first)
; Y Year
; I or l Hour, 12 hour clock
; H Hour, 24 hour clock (single digit hours preceded by "oh")
; k Hour, 24 hour clock (single digit hours NOT preceded by "oh")
; M Minute, with 00 pronounced as "o'clock"
; N Minute, with 00 pronounced as "hundred" (US military time)
; P or p AM or PM
; Q "today", "yesterday" or ABdY
; (*note: not standard strftime value)
; q " (for today), "yesterday", weekday, or ABdY
; (*note: not standard strftime value)
; R 24 hour time, including minute
;
eastern=America/New_York|'vm-received' Q 'digits/at' IMp
central=America/Chicago|'vm-received' Q 'digits/at' IMp
central24=America/Chicago|'vm-received' q 'digits/at' H N 'hours'
military=Zulu|'vm-received' q 'digits/at' H N 'hours' 'phonetic/z_p'
european=Europe/Copenhagen|'vm-received' a d b 'digits/at' HM
```

```
UK24=Europe/London|'vm-received' q 'digits/at' H N 'hours'
UK12=Europe/London|'vm-received' Q 'digits/at' IMp
```

These zones not only specify a time, but also dictate the way times and dates are ordered and read out.

Having created these zones, we can go to the voicemail context part of *voicemail.conf* to associate the appropriate mailboxes with the correct zones:

```
[default]
4001 => 1234,Russell Bryant,rb@shifteight.org,,|tz=central
4002 => 4444,David Duffett,dd@shifteight.org,,|tz=UK24
4003 => 4450,Mary Poppins,mp@shifteight.org,,|tz=UK12|attach=yes
```

As you can see, when we declare a mailbox, we also (optionally) associate it with a particular zone. Full details on voicemail can be found in Chapter 8.

The last thing to localize in our Asterisk configuration is the tones played to callers by Asterisk once they are inside the system (e.g., the tones a caller hears during a transfer).

As identified earlier in this chapter, the initial tones that people hear when they are calling into the system will come from the IP phone, or from DAHDI for analog channels.

These tones are set in */etc/asterisk/indications.conf*. Here is a part of the sample file, where you can see a given region specified by the `country` directive. We just need to change the country code as appropriate:

```
;
; indications.conf
;
; Configuration file for location specific tone indications
;
; NOTE:
; When adding countries to this file, please keep them in alphabetical
; order according to the 2-character country codes!
;
; The [general] category is for certain global variables.
; All other categories are interpreted as location specific indications
;
[general]
country=uk   ; default is US, so we have changed it to UK
```

Your dialplan will need to reflect the numbering scheme for your region. If you do not already know the scheme for your area, your local telecoms regulator will usually be able to supply details of the plan. Also, the example dialplan in */etc/asterisk/extensions.conf* is, of course, packed with North American numbers and patterns.

Conclusion—Easy Reference Cheat Sheet

As you can now see, there are quite a few things to change in order to fully localize your Asterisk-based telephone system, and not all of them are in the Asterisk, or even DAHDI, configuration—some things need to be changed on the connected IP phones or ATAs themselves.

Before we leave the chapter, have a look at Table 9-1: a cheat sheet for what to change and where to change it, for your future reference.

Table 9-1. Internationalization cheat sheet

What to change	Where to change it
Call progress tones	• IP phones—on the phone itself • ATAs—on the ATA itself • Analog phones—DAHDI (*/etc/dahdi/system.conf*)
Type of PRI/BRI and protocol	DAHDI—*/etc/dahdi/system.conf* and */etc/asterisk/chan_dahdi.conf*
Physical PSTN connections	• Balun if required for PRI • Get the analog pair to middle two pins of the RJ11 connecting to the Digium card
Caller ID on analog circuits	Asterisk—*/etc/asterisk/chan_dahdi.conf*
Prompt language and/or accent	• Channel—*/etc/asterisk/sip.conf*, */etc/asterisk/iax.conf*, */etc/asterisk/chan_dahdi.conf*, etc. • Dialplan—`CHANNEL(language)` function
Voicemail time/date stamps and pronunciation	Asterisk—*/etc/asterisk/voicemail.conf*
Tones delivered by Asterisk	Asterisk—*/etc/asterisk/indications.conf*

May all your Asterisk deployments feel at home…

CHAPTER 10
Deeper into the Dialplan

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

Alrighty. You've got the basics of dialplans down, but you know there's more to come. If you don't have Chapter 6 sorted out yet, please go back and give it another read. We're about to get into more advanced topics.

Expressions and Variable Manipulation

As we begin our dive into the deeper aspects of dialplans, it is time to introduce you to a few tools that will greatly add to the power you can exercise in your dialplan. These constructs add incredible intelligence to your dialplan by enabling it to make decisions based on different criteria you define. Put on your thinking cap, and let's get started.

Throughout this chapter we use best practices that have been developed over the years in dialplan creation. The primary one you'll notice is that all the first priorities start with the `NoOp()` application (which simply means No Operation; nothing functional will happen). The other one is that all following lines will start with `same => n`, which is a shortcut that says, "Use the same extension as was just previously defined." Additionally, the indentation is two spaces.

Basic Expressions

Expressions are combinations of variables, operators, and values that you string together to produce a result. An expression can test values, alter strings, or perform

mathematical calculations. Let's say we have a variable called `COUNT`. In plain English, two expressions using that variable might be [`COUNT` plus 1], or [`COUNT` divided by 2]. Each of these expressions has a particular result or value, depending on the value of the given variable.

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

```
$[expression]
```

Thus, we would write our two examples like this:

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

When Asterisk encounters an expression in a dialplan, it replaces the entire expression with the resulting value. It is important to note that this takes place *after* variable substitution. To demonstrate, let's look at the following code:[1]

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

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

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

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

   ```
   same => n,Set(NEWCOUNT=$[3 + 1])
   ```

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

   ```
   same => n,Set(NEWCOUNT=4)
   ```

3. The `Set()` application assigns the value `4` to the `NEWCOUNT` variable.

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

Try it out in your own dialplan.

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

Operators

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

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

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

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

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

Mathematical operators
: Want to perform a calculation? You'll want one of these:

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

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

Regular expression operator
: You can also use the regular expression operator in Asterisk:

Some additional information about the peculiarities of the regular expression operator in Asterisk can be found at Walter Doekes's website.

expr1 : *expr2*
: This operator matches *expr1* against *expr2*, where *expr2* must be a regular expression.[2] The regular expression is anchored to the beginning of the string with an implicit ^.[3]

  If the pattern contains no subexpression, the number of matched characters is returned. This will be `0` if the match failed. If the pattern contains a subexpression `-- \(...\) --` the string corresponding to `\1` is returned. If the match fails, the empty string is returned.

expr1 =~ *expr2*
: This operator works the same as the : operator, except that it is not anchored to the beginning.

Dialplan Functions

Dialplan functions allow you to add more power to your expressions; you can think of them as intelligent variables. Dialplan functions allow you to calculate string lengths, dates and times, MD5 checksums, and so on, all from within a dialplan expression.

You'll see usage of `Playback(silence/1)` throughout the examples in this chapter. We are doing this as it will answer the line if it hasn't already been answered for us, and plays back some silence on the line. This allows other applications such as `SayNumber()` to play back audio without gaps.

Syntax

Dialplan functions have the following basic syntax:

```
FUNCTION_NAME(argument)
```

2 For more on regular expressions, grab a copy of the ultimate reference, Jeffrey E. F. Friedl's *Mastering Regular Expressions* (O'Reilly, 2006), or visit *http://www.regular-expressions.info*.

3 If you don't know what a ^ has to do with regular expressions, you simply must read *Mastering Regular Expressions*. It will change your life!

You reference a function's name the same way as a variable's name, but you reference a function's *value* with the addition of a dollar sign, an opening curly brace, and a closing curly brace:

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

Functions can also encapsulate other functions, like so:

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

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

Examples of Dialplan Functions

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

```
exten => 205,1,Answer()
  same => n,SayDigits(123)
  same => n,SayNumber(123)
  same => n,SayNumber(${LEN(123)})
```

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

```
exten => 206,1,Answer()
  same => n,Set(TIMEOUT(response)=1)
  same => n,Background(enter-ext-of-person)
  same => n,WaitExten()       ; TIMEOUT() has set this to 1
  same => n,Playback(like_to_tell_valid_ext)
  same => n,Set(TIMEOUT(response)=5)
  same => n,Background(enter-ext-of-person)
  same => n,WaitExten()       ; Should be 5 seconds now
  same => n,Playback(like_to_tell_valid_ext)
  same => n,Hangup()
```

Notice the lack of `${ }` surrounding the assignment using the function. Just as if we were assigning a value to a variable, we assign a value to a function without the use of the `${ }` encapsulation; however, if we want to use the value returned by the function, then we need the encapsulation.

```
exten => 207,1,Answer()
  same => n,Set(TIMEOUT(response)=1)
  same => n,SayNumber(${TIMEOUT(response)})
  same => n,Set(TIMEOUT(response)=5)
```

```
same => n,SayNumber(${TIMEOUT(response)})
same => n,Hangup()
```

You can get a list of all active functions with the following CLI command:

```
*CLI> core show functions
```

Or, to see a specific function such as `CALLERID()`, the command is:

```
*CLI> core show function CALLERID
```

Near the end of this chapter, we explore a handful of functions you will want to experiment with. Later in the book we'll show you how to create database-based functions using `func_odbc`.

Conditional Branching

The advanced logic provided through expressions and functions will allow your dialplan to make more powerful decisions, which will often result in *conditional branching*.

The GotoIf() Application

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

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

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

If the expression evaluates to true, the caller is sent to *`destination1`*. If the expression evaluates to false, the caller is sent to the second destination. So, what is true and what is false? An empty string and the number `0` evaluate as false. *Anything else evaluates as true*.

The destinations can each be one of the following:

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

Let's use `GotoIf()` in an example. Here's a little coin toss application. Call it several times to properly test.

```
exten => 209,1,Noop(Test use of conditional branching to labels)
  same => n,GotoIf($[ ${RAND(0,1)} = 1 ]?weasels:iguanas)
; same => n,GotoIf(${RAND(0,1)}?weasels:iguanas) ; works too, but won't in every situation

  same => n(weasels),Playback(weasels-eaten-phonesys) ; NOTE THIS IS SAME EXTENSION
```

```
same => n,Hangup()

same => n(iguanas),Playback(office-iguanas) ; STILL THE SAME EXTENSION
same => n,Hangup()
```

You will notice that we have used the `Hangup()` application following each use of the `Playback()` application. This is done so that when we jump to the `weasels` label, the call stops before execution gets to the *office-iguanas* sound file. It is becoming increasingly common to see extensions broken up into multiple components (protected from each other by the `Hangup()` command), each one a distinct sequence of steps executed following a `GotoIf()`.

Providing Only a False Conditional Path

Either of the destinations may be omitted (but not both). If an expression evaluates to a blank destination, Asterisk simply goes on to the next priority in the current extension.

We could have crafted the preceding example like this:

```
exten => 209,1,Noop(Test use of conditional branching)
  same => n,GotoIf($[ ${RAND(0,1)} = 1 ]?:iguanas)
  same => n,Playback(weasels-eaten-phonesys) ; No weasels label anymore
  same => n,Hangup()
  same => n(iguanas),Playback(office-iguanas) ; NOTE THIS IS THE SAME EXTEN
  same => n,Hangup()
```

There's nothing between the `?` and the `:` so if the statement evaluates to true, execution will continue at the next step. Since that's what we want, a label isn't needed.

We don't really recommend doing this, because it's hard to read. Nevertheless, you will see dialplans like this, so it's good to be aware that this syntax is technically correct.

Rather than using labels, we could also send the call to different extensions. Since they're not dialable, we can use alphabet characters rather than digits for the extension "numbers." In this example, the conditional branch sends the call to completely different extensions within the same context. The result is otherwise the same.

```
exten  => 210,1,Noop(Test use of conditional branching to extensions)
  same => n,GotoIf($[ ${RAND(0,1)} = 1 ]?weasels,1:iguanas,1)

exten => weasels,1,Playback(weasels-eaten-phonesys) ; DIFFERENT EXTENSION
  same => n,Hangup()

exten => iguanas,1,Playback(office-iguanas) ; ALSO A DIFFERENT EXTEN
  same => n,Hangup()
```

Let's look at another example of conditional branching. This time, we'll use both `Goto()` and `GotoIf()` to count down from 5 and then hang up:

```
exten => 211,1,NoOp()
    same => n,Answer()
    same => n,Set(COUNT=5)

    same => n(start),GotoIf($[ ${COUNT} > 0 ]?:goodbye)
    same => n,SayNumber(${COUNT})
    same => n,Set(COUNT=$[ ${COUNT} - 1 ])
    same => n,Goto(start)

    same => n(goodbye),Playback(vm-goodbye)
    same => n,Hangup()
```

Let's analyze this example. In the second priority, we set the variable `COUNT` to `5`. Next, we check to see if `COUNT` is greater than `0`. If it is, we move on to the next priority. (Don't forget that if we omit a destination in the `GotoIf()` application, control goes to the next priority.) From there, we speak the number, subtract `1` from `COUNT`, and go back to priority label `start`. Again, if `COUNT` is less than or equal to `0`, control goes to priority label `goodbye`; otherwise, we run through the loop one more time.

Quoting and Prefixing Variables in Conditional Branches

Now is a good time to take a moment to look at some nitpicky stuff with conditional branches. In Asterisk, it is invalid to have a null value on either side of the comparison operator. Let's look at examples that would produce an error:

```
$[  = 0 ]

$[ foo = ]

$[ > 0 ]

$[ 1 + ]
```

Any of our examples would produce a warning like this:

```
WARNING[28400][C-000000eb]: ast_expr2.fl:470 ast_yyerror: ast_yyerror():
syntax error: syntax error, unexpected '=', expecting $end; Input:
 = 0
 ^
```

It's fairly unlikely (unless you have a typo) that you'd purposefully implement something like our examples. However, when you perform math or a comparison with an unset channel variable, this is effectively what you're doing.

The examples we've used to show you how conditional branching works are not invalid. We've first initialized the variable and can clearly see that the channel variable we're using in our comparison has been set, so we're safe. But what if you're not always so sure?

In Asterisk, strings do not need to be double- or single-quoted like in many programming languages. In fact, if you use a double or single quote, it is a literal construct in the string. If we look at the following snippets of an extension...

```
same => n,Set(TEST_1=foo)
same => n,Set(TEST_2='foo')
same => n,NoOp(Are TEST_1 and TEST_2 equiv?  $[${TEST_1} = ${TEST_2}])
```

...we need to note that the value returned by our comparison in the `NoOp()` will not be a value of `1` (values match; or *true*) the return value will be `0` (values do not match; or *false*).

We can use this to our advantage when performing comparisons by wrapping our channel variables in single or double quotes. By doing this we make sure even when the channel variable might not be set, our comparison will be valid syntax.

In the following example, we would get an error:

```
exten => 212,1,NoOp()
  same => n,GotoIf($[ ${TEST} != valid ]?error_handling)
  same => n,Hangup() ; We're getting an error and ending up here

  same => n(error_handling),Playback(goodbye)
  same => n,Hangup()
```

However, we can circumvent this by wrapping what we're comparing in extra characters (in this case quotes). The same example, but made valid:

```
exten => 213,1,NoOp()
  same => n,GotoIf($[ "${TEST}" != "valid" ]?error_handling)
  same => n,Hangup()

  same => n(error_handling),Playback(goodbye)
  same => n,Hangup()
```

Even if `${TEST}` hasn't been set (in other words it does not exist and therefore has no value), we're still doing a comparison of something:

```
$["" != "valid"]
```

If you get into the habit of recognizing these situations and using the wrapping and prefixing techniques we've outlined, you'll write much safer dialplans.

Note again that the quote character doesn't have any special meaning here. We used it because it's a logical character for this purpose. The following works too:

```
  same => n,GotoIf($[_${TEST}_ != _valid_]?error_handling)
;OR
  same => n,GotoIf($[AAAAA${TEST}AAAAA != AAAAAvalidAAAAA]?error_handling)
```

Not all characters will work, as some may have other meanings to Asterisk and cause problems. Stay with the quote character and you should be fine.

The classic example of conditional branching is affectionately known as the "psycho-ex" logic. If the caller ID number of the incoming call matches the phone number of

somebody you never want to talk to again, Asterisk gives a different message than it ordinarily would to any other caller. While somewhat simple and primitive, it's a good example for learning about conditional branching within the Asterisk dialplan.

This example uses the `CALLERID()` function, which allows us to retrieve the caller ID information on the inbound call. Let's assume for the sake of this example that the victim's phone number is 888-555-1212:[4]

```
exten => 214,1,NoOp(CALLERID(num): ${CALLERID(num)} CALLERID(name): ${CALLERID(name)})
  same => n,GotoIf($[ ${CALLERID(num)} = 8885551212 ]?reject:allow)

  same => n(allow),Dial(${UserA_DeskPhone})
  same => n,Hangup()

  same => n(reject),Playback(abandon-all-hope)
  same => n,Hangup()
```

In priority `1`, we call the `GotoIf()` application. It tells Asterisk to go to priority label `reject` if the caller ID number matches `8885551212`, and otherwise to go to priority label `allow` (we could have simply omitted the label name, causing the `GotoIf()` to fall through).[5] If the caller ID number matches, control of the call goes to priority label `reject`, which plays back a subtle hint to the undesired caller. Otherwise, the call attempts to dial the recipient on the channel referenced by the `UserA_DeskPhone` global variable.

Time-Based Conditional Branching with GotoIfTime()

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

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

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

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

4 If you want to test this (which you do), you can pick one of your working lab devices, and in the `asterisk` database, under the `ps_endpoints` table, set the `callerid` field to `'8885551212'`. Then you can make a call from it to 214 to see the block in action.

```
UPDATE asterisk.ps_endpoints SET callerid='8885551212' WHERE id='<endpoint you chose as the
victim>'
```

5 But we do it this way because it's easier to read.

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

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

It is worth noting that times will properly wrap around. So, if you wish to specify the times your office is closed, you might write `18:00-9:00` in the *`times`* parameter, and it will perform as expected. Note that this technique works as well for the other components of `GotoIfTime()`. For example, you can write `sat-sun` to specify the weekend days.

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

Note that you can specify a combination of ranges and single days, as in: `sun-mon&wed&fri-sat`, or, more simply: `wed&fri-mon`.

`days_of_month`
: This is a list of the numerical days of the month. Days are specified by the numbers `1` through `31`. The 7th through the 12th would be expressed as `7-12`, and the 15th and 30th of the month would be written as `15&30`. This can be useful for holidays, which often fall on the same day of the month, but not typically on the same day of the week.[6]

`months`
: This is a list of one or more months of the year. The months should be written as `jan-apr` for a range, and separated with ampersands when wanting to include nonsequential months, such as `jan&mar&jun`. You can also combine them like so: `jan-apr&jun&oct-dec`.

6 We have no idea how to implement Easter, but are open to suggestions.

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

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

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

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

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

If the caller calls during these hours, the call will be sent to the first priority of the `start` extension in the context named `open`. If the call is made outside of the specified times, it will simply carry on with the next priority of the current extension. We're going to add a new context named `[closed]` right after the pattern match example 55512XX, and modify the `[TestMenu]` context we built in Chapter 6 to handle our new time condition.

```
exten => _55512XX,1,Answer()
  same => n,Playback(tt-monkeys)
  same => n,Hangup()

exten => *98,1,NoOp(Access voicemail retrieval.)
   same => n,VoiceMailMain()

[closed]
exten => start,1,Noop(after hours handler)
  same => n,Playback(go-away2)
  same => n,Hangup()

[TestMenu]
exten => start,1,Noop(main autoattendant)
   same => n,GotoIfTime(16:59-08:00,mon-fri,*,*?closed,start,1)
   same => n,GotoIfTime(11:59-09:00,sat,*,*?closed,start,1)
   same => n,GotoIfTime(00:00-23:59,sun,*,*?closed,start,1)
   same => n,Background(enter-ext-of-person)
   same => n,WaitExten(5)

exten => 1,1,Dial(${UserA_DeskPhone},10)
   same => n,Playback(vm-nobodyavail)
   same => n,Hangup()
```

GoSub

The `GoSub()` dialplan application allows you to send a call off to a separate section of the dialplan, make something useful happen, and then return the call to the point in the dialplan where it came from. You can pass arguments to `GoSub()`, and also receive a return code back from it. This cranks up the functionality of your dialplan quite a bit.

Subroutines are a critical skill in any programming language, and no less so in an Asterisk dialplan. For those new to programming, a subroutine allows you to create a block of generic code that can be reused by different parts of the dialplan to avoid repetition. Think of it like a template in a word processing document, or a blank form, and you've got the general idea. Once you see them in operation, it should become clear how useful they can be.

Defining Subroutines

There are no special naming requirements when using `GoSub()` in the dialplan. In fact, you can use `GoSub()` within the same context and extension if you want to. In most cases, however, your subroutines should be written in separate contexts: one context for each subroutine. When creating the context, we like to prepend the name with `sub` so we know the context is called from the `GoSub()` application.

Let's explore an obvious example of where a subroutine would be useful.

As you might have noticed when we were building our sample dialplan for the users we have created, the dialplan logic for each user can require several lines of code.

```
[sets]
exten => 100,1,Dial(${UserA_DeskPhone},12)
  same => n,Voicemail(100@default)
  same => n,GotoIf($["${DIALSTATUS}" = "BUSY"]?busy:unavail)

  same => n(unavail),VoiceMail(100@default,u)
  same => n,Hangup()

  same => n(busy),VoiceMail(100@default,b)
  same => n,Hangup()

exten => 101,1,Dial(${UserA_SoftPhone})
  same => n,GotoIf($["${DIALSTATUS}" = "BUSY"]?busy:unavail)

  same => n(unavail),VoiceMail(101@default,u)
  same => n,Hangup()

  same => n(busy),VoiceMail(101@default,b)
  same => n,Hangup()

exten => 102,1,Dial(${UserB_DeskPhone},10)
```

```
  same => n,Playback(vm-nobodyavail)
  same => n,Hangup()

exten => 103,1,Dial(${UserB_SoftPhone})
  same => n,Hangup()
```

We've only provided two users with actual, working voicemail, and we've only defined four phones as extensions, and yet we've already got a mess of repetitive code, which is only going to get more and more difficult to maintain and expand. This will quickly become unmanageable if we don't find a better way.

Let's write a subroutine to handle dialing our users. Add the following to the very bottom of your dialplan:

```
; SUBROUTINES
[subDialUser]
exten => _[0-9].,1,Noop(Dial extension ${EXTEN},channel: ${ARG1},  mailbox: ${ARG2})
  same => n,Noop(mboxcontext: ${ARG3}, timeout ${ARG4})
  same => n,Dial(${ARG1},${ARG4})
  same => n,GotoIf($["${DIALSTATUS}" = "BUSY"]?busy:unavail)

  same => n(unavail),VoiceMail(${ARG2}@${ARG3},u)
  same => n,Hangup()

  same => n(busy),VoiceMail(${ARG2}@${ARG3},b)
  same => n,Hangup()
```

Now, modify the top of your dialplan as follows:

```
[OLD_sets] ; what was [sets] is now [OLD_sets] (call it whatever, so long as name changes)
exten => 100,1,Dial(${UserA_DeskPhone},12)
  same => n,Voicemail(100@default)
  same => n,GotoIf($["${DIALSTATUS}" = "BUSY"]?busy:unavail)
;(etc)
```

We've renamed our `[sets]` context, which of course breaks our dialplan since our phones enter the dialplan there. So, we're going to reinsert it a little farther down:

```
exten => 103,1,Dial(${UserB_SoftPhone})
  same => n,Hangup()

[sets]

exten => 110,1,Dial(${UserA_DeskPhone}&${UserA_SoftPhone}&${UserB_SoftPhone})
  same => n,Hangup()
;(etc)
```

OK, so now we've got our `[sets]` context working again, and also this `[OLD_sets]` context that's got our old, orphaned code. How do we dial our telephones? How does this subroutine we just wrote help us?

```
exten => 103,1,Dial(${UserB_SoftPhone})
  same => n,Hangup()

[sets]

;subDialUser args:
```

```
; - ARG1 Channel(s) to dial
; - ARG2 Mailbox
; - ARG3 Mailbox Context
; - ARG4 Timeout
exten => 100,1,Gosub(subDialUser,${EXTEN},1(${UserA_DeskPhone},${EXTEN},default,12))
exten => 101,1,Gosub(subDialUser,${EXTEN},1(${UserA_SoftPhone},${EXTEN},default,3))
exten => 102,1,Gosub(subDialUser,${EXTEN},1(${UserB_DeskPhone},${EXTEN},default,6))
exten => 103,1,Gosub(subDialUser,${EXTEN},1(${UserB_SoftPhone},${EXTEN},default,24))

exten => 110,1,Dial(${UserA_DeskPhone}&${UserA_SoftPhone}&${UserB_SoftPhone})
  same => n,Hangup()
```

Plug that in, reload your dialplan, and make some test calls. Play with the parameters and see what changes. Add some mailboxes to your database and see what that does. If you're inspired, write a new `subDialUserNEW` subroutine and see what you can come up with. At this point you can also delete all the code in the `[OLD_sets]` context, since it's now abandoned, but you can also leave it there as it does no harm.

Now, you can add hundreds of extensions, and each one will only use one line of the dialplan.

Whenever you find yourself writing duplicate dialplan code somewhere, stop. It's very likely that it's time to write a subroutine.

Returning from a Subroutine

The `GoSub()` dialplan application does not return automatically once it is done executing. If you're done with the call, you can of course use `Hangup()`; however, if you don't want to disconnect, but rather need to return the call from where it came, you can use the `Return()` application.

Since you can nest subroutines within subroutines and also execute them one after another, as you get into more complex subroutines you will find this an essential capability.

Local Channels

Local channels are a method of executing other areas of the dialplan from the `Dial()` application (as opposed to sending the call out a channel). Think of them as subroutines you can call from within `Dial()`.

They may seem like a bit of a strange concept when you first start using them, but believe us when we tell you they can be the answer to a problem you can't figure out any other way. You will almost certainly want to make use of them when you start writing advanced dialplans. The best way to illustrate the use of local channels is through an example. Let's suppose we have a situation where we need to ring multiple people, but we need to provide delays of different lengths before dialing each of the members. The use of local channels is the solution to the problem.

With the `Dial()` application, you can certainly ring multiple endpoints (see extension 110 in your dialplan for an example of this), but all three channels will ring at the same time, and for the same length of time.

```
exten => 110,1,Dial(${UserA_DeskPhone}&${UserA_SoftPhone}&${UserB_SoftPhone})
  same => n,Hangup()
```

However, let's say we want to introduce some delays prior to ringing a user, and also stop ringing locations at different times. Using local channels gives us independent control over each of the channels we want to dial, so we can introduce delays and control the period of time for which each channel rings independently.

Let's say we have a small company, where the receptionist is primarily responsible for the incoming calls, but there are also two team members who are tasked with backing up reception, and finally the owner wants to help out if need be too.

These are the requirements:

- The reception phone should ring right away, and keep ringing and not stop until answered.
- The team member phones shouldn't ring for the first 9 seconds, at which point they can ring until answered.
- The owner's phone should only ring if the call has gone on for 12 seconds with no answer. Also, we're pretending it's a cell phone, and thus should stop ringing 18 seconds later so that the call is not answered by the cell phone voicemail.

We'll use our existing configured channels to play the various roles. If you have any way to do so, please try to have them all registered somewhere so they can all ring when called. It'll give you a much better idea of what's going on when testing.[7]

This is a great time for a subroutine:

```
[subDialDelay]
exten => _[a-zA-Z0-9].,1,Noop(channel ${ARG1}, pre-delay ${ARG2}, timeout ${ARG3})
;  same => n,Progress() ; Optional; Signals back that the call is proceeding
  same => n,Wait(${ARG2}) ; how long to wait before dialing
  same => n,Dial(${ARG1},${ARG3}) ; timeout can be blank (infinite)
  same => n,Hangup()
```

You already have a subroutine at the bottom of the file. Add this one down there too so all your subroutines end up grouped together.

7 Obsolete Android phones and tablets can be great for this.

Now we want a context in which we'll build out the extensions to be used by the local channel:

```
;LOCAL CHANNELS
[localDialDelay]
exten => receptionist,1,Gosub(subDialDelay,${EXTEN},1(${UserA_DeskPhone},0,600))
exten => team_one,1,Gosub(subDialDelay,${EXTEN},1(${UserA_SoftPhone},9,600))
exten => team_two,1,Gosub(subDialDelay,${EXTEN},1(${UserB_DeskPhone},9,600))
exten => owner,1,Gosub(subDialDelay,${EXTEN},1(${UserB_SoftPhone},12,18))
```

Even though the destination for a local channel is really just dialplan—the same as you might jump to with a `Goto()`—these constructs tend to be very special-purpose, and fit into the dialplan better in their own area, down with the subroutines. That's why we named the context with the prefix `local`. It's not required, but makes things easier to make sense of.

Now we stitch it all together in our `[sets]` context.

First, let's provide a way to dial each local channel individually, so we can sanity check each one to be sure it's doing what it should.

```
exten => 103,1,Gosub(subDialUser,${EXTEN},1(${UserB_SoftPhone},${EXTEN},default,24))

; These are for testing individually before we put them together
exten => 104,1,Dial(Local/receptionist@localDialDelay)
exten => 105,1,Dial(Local/team_one@localDialDelay)
exten => 106,1,Dial(Local/team_two@localDialDelay)
exten => 107,1,Dial(Local/owner@localDialDelay)
```

Finally, let's deliver the finished product.

```
exten => 107,1,Dial(Local/owner@localDialDelay)

;We're going to assign some variables in order to
;keep the dial string easier to read
exten => 108,1,Noop(DialDelay)
  same => n,Set(Recpn=Local/receptionist@localDialDelay)
  same => n,Set(Team1=Local/team_one@localDialDelay)
  same => n,Set(Team2=Local/team_two@localDialDelay)
  same => n,Set(Boss=Local/owner@localDialDelay)
  same => n,Dial(${Recpn}&${Team1}&${Team2}&${Boss},600)
```

You really need to register a few phones and try this out, to see it all come together.

The solution we have created here is perfect for learning about local channels, but it has a few problems that need to be understood if you ever want to put it into production:

- Even though we have set a dial timeout, you will find that SIP endpoints have minds of their own. It's not uncommon for a SIP endpoint to have its own ideas about timeout. So, you might set it to ring for 600 seconds, and wonder why it drops the call after a minute or so. You could spend hours troubleshooting your dialplan, only to discover the problem was a setting at the other end. *Test each piece of the solution before you glue them all together.*
- Cell phones have their own voicemail, and if that answers the call, Asterisk will connect the call to that "answered" channel. One way around this is to hang up before that happens, and then call immediately back. It's ugly though, and not recommended.
- Cell phones will often go immediately to a voicemail if they're out of range or turned off. *That counts as an answer as far as Asterisk is concerned*. This solution does not handle that.
- Call setup to a cell phone (i.e., the time between when you dial and when it starts ringing) typically takes a dozen seconds or so.
- Remember that a softphone on a cell phone is not at all the same as a phone call to that cell phone. One is a SIP connection, the other is a PSTN call. (You can actually ring both at the same time if you want, but that's not necessarily a good idea.)
- Some types of smartphones will give priority to incoming GSM calls. If you are on a call on the softphone, and somebody calls your cell number, the softphone may get put on hold. Different phones handle this differently.
- We haven't really handled overflow here. What happens if *nobody* answers? It doesn't matter in the lab, but you can be sure it'll matter in a production environment.
- `Dial()` expects ringing back from the destination. If *all* of your local channels have a `Wait()` delay, the caller will hear silence until something indicates ringing. You can fix this by having `Dial()` fake the ringing with the `'r'` option, or by adding a dummy local channel that just returns ringing.

If you check the sample dialplan, we've added a solution to the silence problem on delayed local channels

That's it. Local channels: build them piece-by-piece and you'll be delivering a powerful dialplan in no time.

They're incredibly useful when building complex queueing applications as well.

Using the Asterisk Database

Asterisk provides a simple mechanism for storing data called the *Asterisk database* (AstDB). This is not an external relational database, but simply an SQLite-based backend for storing simple key/value pairs.

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

Storing Data in the AstDB

To store a new value in the Asterisk database, we use the `Set()` application with the `DB()` function. For example, to assign the `count` key in the `test` family with the value of `1`, we would write the following:

```
exten => 216,1,NoOp()
  same => n,Set(DB(testkey/count)=1)
```

Make a test call to 216 to set the value. Note that if a key named `count` already exists in the `test` family, its value will be overwritten with the new value (in this case, the value is hardcoded so obviously will get overwritten with the same value, but later we'll see how we can change the value, and have that stored).

You can also store values from the Asterisk command line, by running the command `database put` *`family key value`*. For our example, you would type `database put test count 1`.

So, while we're at it, let's also plug a value into the database from the console:

```
*CLI> database put somekey somevalue 42
```

Let's query the database from the console to see what values are in there:

```
*CLI> database show
```

If all is well, you should see output similar to the following:

```
/pbx/UUID                                       : d562019a-d2c4-4b88-bcd9-602b3b46fe07
/somekey/count                                  : 1
/somekey/somevalue                              : 42
/testkey/count                                  : 1
4 results found.
localhost*CLI>
```

Retrieving Data from the AstDB

To retrieve a value from the Asterisk database and assign it to a variable, we will again use the `Set()` application and the `DB()` function. Let's retrieve the value of `somevalue` (from the `somekey` family), assign it to a variable called `THE_ANSWER`, and then speak the value to the caller:

```
exten => 217,1,NoOp()
  same => n,Set(THE_ANSWER=${DB(somekey/somevalue)})
  same => n,Answer()
  same => n,SayNumber(${THE_ANSWER})
```

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

Deleting Data from the AstDB

There are two ways to delete data from the Asterisk database. To delete a key, you can use the `DB_DELETE()` application. It takes the path to the key as its arguments, like this:

```
; deletes the key and returns its value in one step
exten => 218,1,Verbose(0, We just blew away ${DB_DELETE(somekey/somevalue)})
```

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

```
exten => 219,1,DBdeltree(somekey)
```

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

If you call extension 217 now, you will see that there is nothing said, because nothing is returned by the database. You can also run `database show` from the CLI, and note that that *`family`* and *`key`* have been removed.

Using the AstDB in the Dialplan

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

To begin the counting example, let's first retrieve a number (the value of the `count` key) from the database and assign it to a variable named `COUNT`. If the key doesn't exist, `DB()` will return `NULL` (no value). Therefore, we can use the `ISNULL()` function

to verify whether or not a value was returned. If not, we will initialize the AstDB with the `Set()` application, where we will set the value in the database to `1`. This will only happen if the database entry does not exist:

```
exten => 220,1,NoOp()
  same => n,Set(COUNT=${DB(test/count)})        ; retrieve current value in database
  same => n,GotoIf($[${ISNULL(${COUNT})}]?firstcount:saycount) ; is there a value?

  same => n(firstcount),Set(DB(test/count)=1)    ; set the value to 1
  same => n,Goto(saycount)

  same => n(saycount),NoOp()
  same => n,Answer
  same => n,SayNumber(${COUNT})
  same => n,Goto(increment)                       ; not reqd but a good habit

  same => n(increment),Set(COUNT=$[${COUNT} + 1]) ; increment by one
  same => n,Set(DB(test/count)=${COUNT})          ; and assign new value to database
  same => n,Goto(saycount)                        ; loop back and say it again
```

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

In the early days of Asterisk, the built-in database was essential. Today, however, it's not as commonly used. It's probably good for setting a few semaphores here and there, but for the most part, if you want to store data, use one of the relational database backends (we discuss relational database integration in later chapters).

Handy Asterisk Features

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

Conferencing with ConfBridge()

The `ConfBridge()` application allows multiple callers to converse together, as if they were all in the same physical location. Some of the main features include:

- The ability to create password-protected conferences
- Conference administration (mute conference, lock conference, or kick off participants)
- The option of muting all but one participant (useful for company announcements, broadcasts, etc.)
- Static or dynamic conference creation
- High-definition audio that can be mixed at sample rates ranging from 8 kHz to 96 kHz

- Video capabilities, including the addition of dynamically switching video feeds based on loudest talker
- Dynamically controlled menu system for both conference administrators and users
- Additional options available in the *confbridge.conf* configuration file

In this chapter we are focused on the dialplan, so we're only going to demonstrate a basic audio conference bridge:

```
$ sudo -u asterisk vim /etc/asterisk/confbridge.conf
[general]

[default_user]
type=user

[default_bridge]
type=bridge
```

After building the *confbridge.conf* file, we need to load the `app_confbridge.so` module. This can be done at the Asterisk console:

```
*CLI> module load app_confbridge.so
```

With the module loaded, we can build a simple dialplan to access our conference bridge:

```
exten => 221,1,NoOp()
    same => n,ConfBridge(${EXTEN})
```

This is just the tip of the iceberg for conferencing. We've got the base configuration done, but there is much more functionality to be configured. We'll cover it in a little more detail in Chapter 11.

Handy Dialplan Functions

We discussed functions earlier in this chapter, but there's more to say. There are currently around 150 dialplan functions provided by the Asterisk dialplan. Here is a small, curated list of a few worth experimenting with.

CALLERID()

`CALLERID()` supports many different datatypes, but you'll find that you'll typically use one of `name` or `num`.

```
exten => 222,1,Noop(CALLERID function)
  same => n,Noop(CALLERID currently ${CALLERID(all)})
  same => n,Set(CALLERID(num)=4169671111)
  same => n,Noop(CALLERID now ${CALLERID(all)})
  same => n,Set(CALLERID(name)="Somename")
```

```
  same => n,Noop(CALLERID now ${CALLERID(all)})
  same => n,Hangup()
```

Don't worry about the rest of them. If you need 'em, you'll know what they are or why you want to use them.

CHANNEL()

`CHANNEL()` allows you to interact with an absolute boatload of data relating to the channel. Some items allow you to modify them, while others will only be useful for reference (for example, `peerip` will allow you to read, but not change, the IP address of the peer). There are also channel variables that only work with certain channel types (for example, `pjsip` items can of course only be used on PJSIP channels).

```
exten => 223,1,Noop(CHANNEL function)
  same => n,Answer()
  same => n,Noop(CHANNEL(name) is ${CHANNEL(name)})
  same => n,Noop(CHANNEL(musicclass) is ${CHANNEL(musicclass)})
  same => n,Noop(CHANNEL(rtcp,all_jitter) is ${CHANNEL(rtcp,all_jitter)})
  same => n,Noop(CHANNEL(rtcp,all_loss) is ${CHANNEL(rtcp,all_loss)})
  same => n,Noop(CHANNEL(rtcp,all_rtt) is ${CHANNEL(rtcp,all_rtt)})
  same => n,Noop(CHANNEL(rtcp,txcount) is ${CHANNEL(rtcp,txcount)})
  same => n,Noop(CHANNEL(rtcp,rxcount) is ${CHANNEL(rtcp,rxcount)})
  same => n,Noop(CHANNEL(pjsip,local_uri) is ${CHANNEL(pjsip,local_uri)})
  same => n,Noop(CHANNEL(pjsip,remote_uri) is ${CHANNEL(pjsip,remote_uri)})
  same => n,Noop(CHANNEL(pjsip,request_uri) is ${CHANNEL(pjsip,request_uri)})
  same => n,Noop(CHANNEL(pjsip,local_tag) is ${CHANNEL(pjsip,local_tag)})
```

CURL()

`CURL()` is a simple yet powerful function that provides a one-liner method for resolving URLs, which in many cases is all you need for a basic interaction with an external web service.

```
exten => 224,1,Noop(CURL function)
  same => n,Set(ExternalIP=${CURL(http://whatismyip.akamai.com)})
  same => n,Noop(The external IP address is ${ExternalIP})
```

If you need a more complex interaction with an external service, it could be that you are going to want an AGI program of some sort. Still, you can embed a ton of data in a URL, and for simplicity, `CURL()` is hard to beat.

CUT()

If you need to slice-and-dice your variables, you'll find the `CUT()` function essential. The form is simple:

```
CUT(varname,char-delim,range-spec)
```

It can be visually tricky, as the delimiter character can be difficult to see nested in between two commas (for example, if the delimiter was a dot/decimal/period). Let's

expand on the previous example to see what it's good for (and give you a visual example of how the delimiter can get lost in the syntax).

```
exten => 225,1,Noop(CUT function)
  same => n,Set(ExternalIP=${CURL(http://whatismyip.akamai.com)})
  same => n,Noop(The external IP address is ${ExternalIP})
  same => n,Answer()
  same => n,SayDigits(=${CUT(ExternalIP,.,1)})
  same => n,Playback(letters/dot)
  same => n,SayDigits(=${CUT(ExternalIP,.,2)})
  same => n,Playback(letters/dot)
  same => n,SayDigits(=${CUT(ExternalIP,.,3)})
  same => n,Playback(letters/dot)
  same => n,SayDigits(=${CUT(ExternalIP,.,4)})
```

Note that you call the `CUT()` function with the braces `${CUT()}`, but the variable being referenced inside `CUT()` is defined without the braces. This is because we are naming the variable, not asking for its contents (`CUT()` will deal with the contents, so we just need to name the variable it will be slicing and dicing, and it will dive into what is stored there).

IF() and STRFTIME()

The combination of `IF()` and `STRFTIME()` is a powerful construct, and you will find it an essential part of your dialplan logic:

```
exten => 226,1,Noop(IF)
  same => n,Answer()
  same => n,Playback(${IF($[$[${STRFTIME(,,%S)} % 2] = 1]?hear-odd-noise:good-evening)})
```

Wait...what?[8]

Let's break this down (we're going to indent the code in an odd manner in order to show the progression of the nested functions and operators):

```
exten => 227,1,Noop(IF)
  same => n,Answer()
  same => n,Wait(.5)
                same => n,Wait(.5)
              same => n,Noop(${STRFTIME(,,%S)})       ; current time - just seconds
            same => n,Noop($[ ${STRFTIME(,,%S)} % 2 ]) ; divide by 2 - return remainder
    same => n,Noop(${IF($[ $[ ${STRFTIME(,,%S)} % 2 ] = 1 ]?odd:even)})
same => n,Playback(${IF($[ $[ ${STRFTIME(,,%S)} % 2 ] = 1 ]?hear-odd-noise:good-evening)})
```

8 There is a C language function named `STRFTIME()` that returns the current time as a formatted string. This works similarly to that. In fact, the `format` portion of the function takes the exact same syntax as the C function.

The IF() function allows us to pass logic to the Playback() application. We're effectively saying, "If it's true that the time, in seconds, is odd, play the hear-odd-noise prompt, otherwise, play the good-evening prompt."

If we line up the code in a more typical fashion, it looks like this (note that some of the optional spaces have also been removed):

```
exten => 228,1,Noop(IF)
  same => n,Answer()
  same => n,Wait(.5)
  same => n,Noop(${STRFTIME(,,%S)})
  same => n,Noop($[${STRFTIME(,,%S)} % 2])
  same => n,Noop(${IF($[$[${STRFTIME(,,%S)} % 2 ] = 1]?odd:even)})
  same => n,Playback(${IF($[$[${STRFTIME(,,%S)} % 2 ] = 1]?hear-odd-noise:good-evening)})
```

The final line is very difficult to read unless you know how we got there, but it demonstrates the power of nesting.

At first these constructs may seem difficult to write, so break them down and perform them line by line, and eventually they'll get easier (and your dialplan will subsequently become more powerful). Play with them.

LEN()

Being able to return the length of something with the LEN() function can be very handy.

```
exten => 229,1,Noop(LEN)
  same => n,Set(LengthyString=${RAND(1,2000)})
  same => n,Noop(${LEN(${LengthyString})})
  same => n,Noop(${IF( $[ ${LEN(${LengthyString})} <= 3 ]?tooshort:youcanride)})
```

REGEX()

Yes, you can use regular expressions within Asterisk. This is a somewhat advanced topic, not because REGEX() is a complicated function in itself, but because regular expressions are a study in themselves.

Check out *http://www.regular-expressions.info/* for more info, or grab a copy of O'Reilly's *Mastering Regular Expressions* by Jeffrey E. F. Friedl.

Get used to using other functions in Asterisk, get some experience with regular expressions, and then give REGEX() a try.

STRFTIME()

We just saw the STRFTIME() function in our IF() example. It allows you to return a time in various formats. In general, you want the input to be empty (which defaults to the current time). You can also give this function a specific Unix epoch string and it'll work from that.

```
exten => 230,1,Noop(STRFTIME)
  same => n,Noop(${STRFTIME(,,%S)}) ; we've seen this before
  same => n,Noop(${STRFTIME(,,%B)}) ; month
  same => n,Noop(${STRFTIME(,,%H)}) ; hour in 24hr format
  same => n,Noop(${STRFTIME(,,%m)}) ; month as a decimal
  same => n,Noop(${STRFTIME(,,%M)}) ; minute
  same => n,Noop(${STRFTIME(,,%Y)}) ; year - 4 digits
  same => n,Noop(${STRFTIME(,,%Y-%m-%d %H:%m:%S)}) ; string some together
```

Conclusion

In this chapter, we've covered a few more of the many applications in the Asterisk dialplan, and hopefully we've given you some more tools that you can use to further experiment with creating your own dialplans. As with other chapters, we invite you to go back and reread any sections that require clarification.

CHAPTER 11

PBX Features, Including Parking, Paging, and Conferencing

I don't believe in angels, no. But I do have a wee parking angel. It's on my dashboard and you wind it up. The wings flap and it's supposed to give you a parking space. It's worked so far.

—Billy Connolly

This chapter discusses several peripheral features common to business telephone environments. We'll briefly cover the *features.conf* file, and then spend a few sections on paging and parking, and finally do a bit of work with Asterisk's conferencing engine, `confbridge`.

First up, let's copy the *features.conf* file over from the installation directory, and have a look at it:

```
$ sudo cp ~/src/asterisk-16.<TAB>/configs/samples/features.conf.sample \
/etc/asterisk/features.conf

$ sudo chown asterisk:asterisk /etc/asterisk/features.conf
```

features.conf

Asterisk provides several features common to most PBXs, many of which have optional parameters. The *features.conf* file is where you can adjust or define the various feature parameters in Asterisk.

DTMF-Based Features

Many of the parameters in *features.conf* only apply when invoked on calls that have been bridged by the dialplan applications `Dial()` or `Queue()` using one or more of the options `K`, `k`, `H`, `h`, `T`, `t`, `W`, `w`, `X`, or `x`. Features accessed in this way are DTMF-based (meaning they can't be accessed via SIP messaging, but only through touch-tone signals in the audio channel triggered by users dialing the required digits on their dialpads).[1]

Transfers on SIP channels (for example, from a SIP telephone) can be handled using the capabilities of the phone itself and won't be affected by anything in the *features.conf* file.

The [general] Section

In the `[general]` section of *features.conf*, you can define options that fine-tune the behavior of the transfers feature in Asterisk. These have nothing to do with how SIP telephones handle call transfers. You instead access these features by using DTMF digits while on a call (the call must be bridged, so calls ringing or in progress will not have access to these features).

The *features.conf.sample* file in your *~/asterisk/* folder contains details of the various options, and examples of how they can be set.

These features are not as commonly used as they were in the past, mostly because many of these things can be handled in more advanced ways than firing DTMF from the telephone set (for example, through an external integration of some sort, or for that matter from the telephone itself using its own internal transfer features).

The [featuremap] Section

The `[featuremap]` section, summarized in Table 11-1, allows you to define specific DTMF sequences that will trigger features on channels that have been bridged via options in the `Dial()` or `Queue()` application. The two options you are most likely to use are `parkcall` and `automixmon`.

1 Yes, we realize that a SIP INFO message is in fact a SIP message and not technically part of the audio channel, but the point is that you can't use the "transfer" or "park" button on your SIP phone to access these features while on a call. You'll have to send DTMF.

Table 11-1. features.conf [featuremap] section

Option	Value/example	Notes	Dial()/Queue() flags
`blindxfer`	`#1`	Invokes a blind (unsupervised) transfer	`T, t`
`disconnect`	`*0`	Hangs up the call	`H, h`
`automon`	`*1`	Starts recording of the current call using the `Monitor()` application (pressing this key sequence a second time stops the recording)	`W, w`
`atxfer`	`*2`	Performs an automated transfer	`T, t`
`parkcall`	`#72`	Parks a call	`K, k`
`automixmon`	`*3`	Starts recording of the current call using the `MixMonitor()` application (pressing this key sequence again stops the recording)	`X, x`

The [applicationmap] Section

The `[applicationmap]` section of *features.conf* is arguably the most nifty, as it allows you to map DTMF codes to dialplan applications. The caller will be placed on hold until the application has completed execution.

The syntax for defining an application map is as follows (it must appear on a single line; line breaks are not allowed):[2]

```
Name => DTMF_sequence,ActivateOn[/ActivatedBy],App([Args])[,MOH_Class]
```

What you are doing is the following:

1. Giving your map a name so that it can be enabled in the dialplan through the use of the `DYNAMIC_FEATURES` channel variable (more on this in a moment).
2. Defining the DTMF sequence that activates this feature (we recommend using at least two digits for this).
3. Defining which channel the feature will be activated on, and (optionally) which participant is allowed to activate the feature (the default is to allow both channels to use/activate this feature).
4. Giving the name of the application that this map will trigger, and its arguments.
5. Providing an optional music on hold (MOH) class to assign to this feature (which the opposite channel will hear when the application is executing). If you do not define any MOH class, the caller will hear only silence.

2 There is some flexibility in the syntax (you can look at the sample file for details), but our example uses the style we recommend, since it's the most consistent with typical dialplan syntax.

Here is an example of an application map that will trigger an AGI script:[3]

```
agi_test => *6,self/callee,AGI(agi-test.agi),default
```

You may add this to your */etc/asterisk/features.conf* file if you wish.

Since applications spawned from the application map are run outside the PBX core, you cannot execute any applications that trigger the dialplan (such as `Goto()`, `Macro()`, `Background()`, etc.). If you wish to use the application map to spawn external processes (including executing dialplan code), you will need to trigger an external application through an `AGI()` call or the `System()` application. The point is, if you want anything complex to happen through the use of an application map, you will need to test very carefully, as not all things will work as you might expect.

To use an application map, you must declare it in the dialplan by setting the `DYNAMIC_FEATURES` variable somewhere before the `Dial()` command that connects the channels. Use the double underscore modifier on the variable name to ensure that the application map is available to both channels throughout the life of the call. Let's toss this one in our `subDialUser` subroutine, so that it'll be available whenever any of our extensions call each other:

```
[subDialUser]
exten => _[0-9].,1,Noop(Dial extension ${EXTEN},channel: ${ARG1},  mailbox: ${ARG2})
  same => n,Noop(mboxcontext: ${ARG3}, timeout ${ARG4})
  same => n,Set(__DYNAMIC_FEATURES=agi_test)
  same => n,Dial(${ARG1},${ARG4})
  same => n,GotoIf($["${DIALSTATUS}" = "BUSY"]?busy:unavail)
```

If you want to allow more than one application map to be available on a call, you will need to use the `#` symbol as a delimiter between multiple map names:

```
Set(__DYNAMIC_FEATURES=agi_test#my_other_map)
```

The reason why the `#` character was chosen instead of a simple comma is that older versions of the `Set()` application interpreted the comma differently than more recent versions, and the syntax for application maps has never been updated.

Don't forget to reload the `features` module after making changes to the *features.conf* file:

3 We'll cover AGI in Chapter 18, but briefly, AGI scripts are external programs you can trigger from the dialplan. Handy? Very!

```
*CLI> module reload features
```

You can verify that your changes have taken place through the CLI command `features show`.

Also, since we are introducing an AGI script here, there are some commands that must be run to make the referenced AGI script available to Asterisk.

```
$ sudo cp ~/src/asterisk-16.<TAB>/agi/agi-test.agi /var/lib/asterisk/agi-bin/

$ sudo chown asterisk:asterisk /var/lib/asterisk/agi-bin/*

$ sudo chmod 755 /var/lib/asterisk/agi-bin/*
```

Make sure you test out your application map before you turn it over to your users!

Dynamic Feature-Map Creation from the Dialplan

You can create feature maps from the dialplan directly, making the definition of a feature (and its DTMF mapping) dynamic, on a per-channel basis. This is done with the `FEATURE()` and `FEATUREMAP()` dialplan functions. Valid values for `FEATUREMAP()` include the following, which set or retrieve the DTMF sequence used to trigger the functionality:

`atxfer`
: Attended transfer

`blindxfer`
: Blind transfer

`automon`
: Auto `Monitor()` (call recording)

`disconnect`
: Call disconnect

`parkcall`
: Call parking

`automixmon`
: Auto `MixMonitor()` (call recording)

With `FEATUREMAP()`, the function can be used to retrieve the current DTMF sequence for that functionality:

```
exten => 232,1,Noop(Current DTMF for parkcall: ${FEATUREMAP(parkcall)})
```

Or you can use the DTMF sequence for a feature function on the current channel:

```
exten => 233,1,NoOp()
    same => n,Set(FEATUREMAP(parkcall)=*9)
    same => n,Noop(DTMF for parkcall now: ${FEATUREMAP(parkcall)})
```

If you want to set the parking timeout for a channel, you can do so with the `FEATURE()` function. It contains a single argument, `parkingtime`, which is a value in seconds before the parked call is returned to the caller (or destination, depending on how you've configured parking):

```
exten => 234,1,NoOp()
    same => n,Set(FEATURE(parkingtime)=60)
```

Application Map Grouping

If you have a lot of features that you need to activate for a particular context or extension, you can group several features together in an application map grouping, so that one assignment of the `DYNAMIC_FEATURES` variable will assign all of the designated features of that map.

The application map groupings are added at the end of the *features.conf* file. Each grouping is given a name, and then the relevant features are listed:

```
[shifteight]
unpauseMonitor => *1    ; custom key mapping
pauseMonitor => *2      ; custom key mapping
agi_test =>             ; no custom key mapping
```

If you want to specify a custom key mapping to a feature in an application map grouping, simply follow the `=>` with the key mapping you want. If you do not specify a key mapping, the default key map for that feature will be used (as found in the `[featuremap]` section). Regardless of whether you want to assign a custom key mapping or not, the `=>` operator is required.

In the dialplan, you would assign this application map grouping with the `Set()` application:

```
same => Set(__DYNAMIC_FEATURES=shifteight) ; use the double underscore if you want
                                           ; to ensure both call legs have the
                                           ; variable assigned.
```

Parking and Paging

Although these two features are completely separate from each other, they are so commonly used together we might as well treat them as one single feature.

Call parking allows calls to be placed on hold and then retrieved from a location different from where they were originally answered. Paging uses a public address system to allow announcements to be sent from the telephone system (for example, to announce who the parked call is for and how it can be retrieved).

Some businesses, perhaps with large warehouses, outdoor areas, or employees who move around the office a lot, utilize the paging and parking functionality of their systems to direct calls around the office. In this chapter we'll show you how to use both parking and paging in the traditional setting (park 'n' page), along with a couple of more modern takes on these commonly used functions.

Call Parking

A parking lot allows a call to be held in the system without being associated with a particular extension. The call can then be retrieved by anyone who knows the park code for that call. This feature is often used in conjunction with a public address (PA), or "paging" system. For this reason, it is often referred to as "park-and-page." It should be noted that parking and paging are in fact separate. We'll cover paging momentarily, but first, let's talk about call parking.

Let's grab a copy of the sample file that we'll use to configure call parking:

```
$ sudo cp ~/src/asterisk-16.<TAB>/configs/samples/res_parking.conf.sample \
/etc/asterisk/res_parking.conf

$ sudo chown asterisk:asterisk /etc/asterisk/res_parking.conf

$ sudo asterisk -rx 'module load res_parking.so'
```

To park a call in Asterisk, you need to transfer the caller to the feature code assigned to parking, which is assigned in the *res_parking.conf* file with the `parkext` directive. By default, this is `700`:

```
parkext => 700      ; What extension to dial to park (all parking lots)
```

You have to wait to complete the transfer until you get the number of the parking retrieval slot from the system, or you will have no way of retrieving the call. By default the retrieval slots, assigned with the `parkpos` directive in *res_parking.conf*, are numbered from `701` to `720`:

```
parkpos => 701-720  ; What extensions to park calls on (defafult parking lot)
```

Once the call is parked, anyone on the system can retrieve it by dialing the number of the retrieval slot (`parkpos`) assigned to that call. The call will then be bridged to the channel that dialed the retrieval code.

There are two common ways to define how retrieval slots are assigned. This is done with the `findslot` directive in the *res_parking.conf* file. The default method (`findslot => first`) always uses the lowest-numbered slot if it is available, and only assigns higher-numbered codes if required. The second method (`findslot => next`) will rotate through the retrieval codes with each successive park, returning to the first retrieval code after the last one has been used. Which method you choose will depend on how busy your parking lots are. If you use parking rarely, the default `findslot` of `first` will be best (people will be used to their parked calls always being in the same

slot). If you constantly use the parking feature (for example, in an automobile dealership), it is far better for each successive page to assign the next slot, since you will often have more than one call parked at a time. Your users will get used to listening carefully to the actual parking lot number (instead of just always dialing 701), and this will minimize the chance of people accidentally retrieving the wrong call on a busy system.

Handling Timed-Out Parked Calls with the comebacktoorigin Option

This option configures the behavior of call parking when the parked call times out (see the `parkingtime` option). `comebacktoorigin` can have one of two values:

`yes` *(default)*
: When the parked call timeout is exceeded, Asterisk will attempt to send the call back to the peer that parked this call. If the channel is no longer available to Asterisk, the caller will be disconnected.

`no`
: This option would be used when you want to perform custom dialplan functionality on parked calls that have exceeded their timeouts. The caller will be sent into a specific area of the dialplan where logic can be applied to gracefully handle the remainder of the call (this may involve simply returning the call to a different extension, or performing a lookup of some sort).

You also may need to take into account calls where the originating channel cannot handle a returned parked call. If, for example, the call was parked by a channel that is also a trunk to another system, there would not be enough information to send the call back to the correct person on that other system. The actions following a timeout would be more complex than `comebacktoorigin=yes` could handle gracefully.

Parked calls that timeout with `comebacktoorigin=no` will always be sent into the `parkedcallstimeout` context.

The dialplan (and contexts) were discussed in detail in Chapter 6.

The extension they will be sent to will be built from the name of the channel that parked the call. For example, if a SIP peer named `0004F2040808` parked this call, the extension will be `SIP_0004F2040808`.

If this extension does not exist, the call will be sent to the `s` extension in the `parkedcallstimeout` context instead. Finally, if the `s` extension of `parkedcallstimeout` does not exist, the call will be sent to the `s` extension of the `default` context.

Additionally, for any calls where `comebacktoorigin=no`, there will be an extension of `SIP_0004F2040808` created in the `park-dial` context. This extension will be set up to do a `Dial()` to `SIP/0004F2040808`.[4]

If you are using parking, you are also going to need a way to announce the parked calls so users know how to retrieve them. While you could just run down the hall yelling "Bob, there's a call for you on 701!," the more professional method is to use a paging system (more formally known as a public address system), which we will discuss now.

Paging (aka Public Address)

In many PBX systems, it is useful to connect the telephone system to some sort of public address system. This involves dialing a feature code or extension that makes a connection to a public address resource of some kind, and then making an announcement through the handset of the telephone that is broadcast to all devices associated with that paging resource (perhaps you've heard the clerk at your grocery store request a price check through the telephone). Often, this will be an external paging system consisting of an amplifier connected to overhead speakers; however, paging through the speakers of office telephones is also popular (mainly for cost reasons). If you have the budget (or an existing overhead paging system), overhead paging is generally better, but set-based paging can work well in many environments. It is also possible to have a mix of set-based and overhead paging, where, for example, set-based paging might be in use for the administrative offices, but overhead paging would be used for warehouse, hallway, parking lot, and public areas (cafeteria, reception, etc.).

In Asterisk, the `Page()` application is used for paging. This application simply takes a list of channels as its argument, calls all of the listed channels simultaneously, and, as they are answered, puts each one into a conference room. With this in mind, it becomes obvious that one requirement for paging to work is that *each destination channel must be able to automatically answer the incoming connection* and place the call onto a speaker of some sort (in other words, `Page()` won't work if all the phones just ring).

So, while the `Page()` application itself is painless and simple to use, getting all the destination channels to handle the incoming pages correctly is a bit trickier. We'll get to that shortly.

4 We hope you realize that the actual extension will be related to the channel name that parked the call, and will not be `SIP_0004F2040808` (unless Leif sells you the Polycom phone from his lab).

The `Page()` application takes three arguments: 1) the group of channels the page is to be connected to, 2) the options, and 3) the timeout:

```
exten => *724,1,Noop(Page)
  same => n,Set(ChannelsToPage=${UserA_DeskPhone}&${UserA_SoftPhone}}&${UserB_DeskPhone})
  same => n,Page(${ChannelsToPage},i,120)
```

The options (outlined in Table 11-2) give you some flexibility with respect to how `Page()` works, but the majority of the configuration is going to have to do with how the target devices handle the incoming connection. We'll dive into the various ways you can configure devices to receive pages in the next section.

Table 11-2. Page() options

Option	Description	Discussion
`d`	Enables full-duplex audio	Sometimes referred to as "talkback paging," the use of this option implies that the equipment that receives the page has the ability to transmit audio back at the same time as it is receiving audio. Generally, you don't want to use this unless you have a specific need for it.
`i`	Ignores attempts to forward the call	You would normally want this option enabled, because a call-forwarded set could go pretty much anywhere, and that's not where your page needs to go.
`q`	Does not play beep to caller (quiet mode)	Normally you won't use this, since it's good for paging to make a sound to alert people that a page is about to happen. However, if you have an external amplifier that provides its own tone, you may want to set this option.
`r`	Records the page into a file	If you intended to use the same page multiple times in the future, you could record the page and then use it again later by triggering it using `Originate()` or using the `A(x)` option to `Page()`.
`s`	Dials a channel only if the device state is `NOT_INUSE`	This option is likely only useful (and reliable) on SIP-bound channels, and even so may not work if a single line is allowed to host multiple calls simultaneously (quite common with SIP phones). Therefore, *don't rely on this option* in all cases.
`A(x)`	Plays announcement `x` to all participants	You could use a previously recorded file to be played over the paging system. If you combined this with `Originate()` and `Record()`, you could implement a delayed paging system.
`n`	Does not play announcement simultaneously to caller (implies `A(x)`)	By default, the system will play the paged audio to both the caller and the callee. If this option is enabled, the paged audio will not be played to the caller (the person paging).

Because of how `Page()` works, it is very resource-intensive. We cannot stress this enough. Carefully read on, and we'll cover how to ensure that paging does not cause performance problems in a production environment (which it is almost certain to do if not designed correctly).

Places to Send Your Pages

As we stated before, `Page()` is, in and of itself, very simple. The trick is how to bring it all together. Pages can be sent to different kinds of channels, and they all require different configuration.

External paging

If a public address system is installed in the building, it is common to connect the telephone system to an external amplifier and send pages to it through a call to a channel. The best way we know of doing this is to use an FXS device of some kind (such as an ATA), which is connected through a paging interface such as a Bogen UTI1,[5] which then connects to the paging amplifier.[6]

Another popular way to connect to a paging system is to plug the output of the sound card of your Asterisk server into the paging amplifier, and send calls to the channel named `Console/DSP`. We don't like this method because while it may seem inexpensive and simple, in practice it can be time-consuming. It asumes that the sound drivers on your server are working correctly, the audio levels are normalized correctly on that channel, your server has a decent on-board audio card, the grounding is good, and...well, in our opinion, this way is not recommended.[7]

In your dialplan, paging to an external amplifier would look similar to a simple `Dial()` to the device that is connected to the paging equipment. You would need to configure the ATA the same as any SIP telephone (through `ps_endpoints`, `ps_auth`, etc., in the database), named something like `PagingATA`. You then plug the ATA into a Bogen UTI1, and to page you have this dialplan code:

5 The Bogen UTI1 is useful because it can handle all manner of incoming and outgoing connections, which pretty nearly guarantees that you'll be able to painlessly connect your telephone system to any sort of external paging equipment, no matter how old or obscure. The cost of the unit can be offset against the hours of time saved by having a purpose-built, fully featured, Swiss-army-knife-like interface to an existing public address system (or, for that matter, as part of a new PA system).

6 In this book we're assuming that the external paging equipment is already installed and was working with the old phone system, but there's nothing stopping you from installing a brand-new public address system and connecting it to your Asterisk system. You might feel that we're plugging Bogen a bit here, but it's simply because they have been doing telephone system paging for a very long time. We've been using them for nearly 30 years, and they've been doing it for longer, so as long as you're comfortable putting all the pieces together, you can get the job done right the first time.

7 If you're at all curious about this, we want to encourage you to try this out in your lab. It might prove to work very well for you, and it does potentially save some hardware costs. We've just found that hardware is cheaper than labor, so we'd rather drop a couple of hundred bucks on known-good gear than have some poor technician mucking about onsite for eight hours with an upset customer demanding to know when the paging problem will be solved.

```
exten => *725,1,Verbose(2,Paging to external amplifier) ; The '*' is part of what you dial
    same => n,Set(PageDevice=PJSIP/PagingATA)         ; This probably belongs in [globals]
    same => n,Page(${PageDevice},i,120)
```

You can name this device anything you want (for example, we often use the MAC address as the name of a SIP device), but for anything that is not a user telephone, it can be helpful to use a name that makes it stand out from other devices.

There are also many SIP-based paging devices on the market (SIP paging speakers are popular, but—we think—rather expensive for what you get, especially in a large deployment).

Set paging

Set-based paging first became popular in key telephone systems, where the speakers of the office telephones are used as a pauper's public address system. Most SIP telephones have the ability to auto-answer a call on handsfree, which accomplishes what is required on a per-telephone basis. In addition to this, however, it is necessary to pass the audio to more than one set at the same time. Asterisk uses its built-in conferencing engine to handle the under-the-hood details. You use the `Page()` application to make it happen.

Like `Dial()`, the `Page()` application can handle several channels. Since you will generally want `Page()` to signal several sets at once (perhaps even all the sets on your system), you may end up with lengthy device strings that look something like this:

```
Page(PJSIP/SET1&PJSIP/SET2&PJSIP/SET3&PJSIP/SET4&PJSIP/SET5&PJSIP/SET6&PJSIP/SET7&...
```

Beyond a certain size, your Asterisk system will be unable to page multiple sets. For example, in an office with 200 telephones, using SIP to page every set would not be possible; the traffic and CPU load on your Asterisk server would simply be too much. In cases like this, you should be looking at either multicast paging or external paging.

Perhaps the trickiest part of SIP-based paging is the fact that you usually have to tell each set that it must auto-answer, but different manufacturers of SIP telephones use different SIP messages for this purpose. So, depending on the telephone model you are using, the commands needed to accomplish SIP-based set paging will be different. Here are some examples:

- For Mitel (FKA Aastra):

```
exten => *726,1,Verbose(2,Paging to Aastra sets)
    same => n,SIPAddHeader(Alert-Info: info=alert-autoanswer)
    same => n,Set(PageDevice=SIP/00085D000000)
    same => n,Page(${PageDevice},i)
```

- For Polycom:

```
exten => *727,1,Verbose(2,Paging to Polycom sets)
    same => n,SIPAddHeader(Alert-Info: Ring Answer)
    same => n,Set(PageDevice=SIP/0004F2000000)
    same => n,Page(${PageDevice},i)
```

- For Snom:

```
exten => *728,1,Verbose(2,Paging to Snom sets)
    same => n,Set(VXML_URL=intercom=true)

; replace 'domain.com' with the domain of your system
    same => n,SIPAddHeader(Call-Info: sip:domain.com\;answer-after=0)
    same => n,Set(PageDevice=SIP/000413000000)
    same => n,Page(${PageDevice},i)
```

- For Cisco SPA (the former Linksys phones, not the 79XX series):

```
exten => *729,1,Verbose(2,Paging to Cisco SPA sets, but not Cisco 79XX sets)
    same => n,SIPAddHeader(Call-Info:\;answer-after=0)    ; Cisco SPA phones
    same => n,Set(PageDevice=SIP/0004F2000000)
    same => n,Page(${PageDevice},i)
```

Assuming you've figured that out, what happens if you have a mix of phones in your environment? How do you control which headers to send to which phones?[8]

Any way you slice it, it's not pretty.

Fortunately, many of these sets support IP multicast, which is a far better way to send a page to multiple sets (read on for details). Still, if you only have a few phones on your system and they are all from the same manufacturer, SIP-based paging could be the simplest method, so we don't want to scare you off it completely.

Multicast paging via the MulticastRTP channel

If you are serious about paging through the sets on your system, and you have more than a handful of phones, you will need to look at using IP multicast. The concept of IP multicast has been around for a long time,[9] but it has not been widely used. Nevertheless, it is ideal for paging within a single location.

Asterisk has a channel (`chan_multicast_rtp`) that is designed to create an RTP multicast. This stream is then subscribed to by the various phones, and the result is that whenever media appears on the multicast stream, the phones will pass that media to their speakers.

8 Hint: the local channel will be your friend here.

9 It even has its own Class D reserved IP address space, from 224.0.0.0 to 239.255.255.255 (but read up on IP multicast before you just grab one of these and assign it). Parts of this address space are private, parts are public, and parts are designated for purposes other than what you might want to use them for. For information about multicast addressing, see its Wikipedia page.

Since `MulticastRTP` is a channel driver, it does not have an application, but instead will work anywhere in the dialplan that you might otherwise use a channel. In our case, we'll be using the `Page()` application to initiate our multicast.

To use the multicast channel, you simply send a call to it the same as you would to any other channel. The syntax for the channel is as follows:

```
MulticastRTP/type/ip address:port[/linksys address:port]
```

The type can be either `basic` or `linksys`. The basic syntax of the `MulticastRTP` channel looks like this:

```
exten => *730,1,Page(MulticastRTP/basic/239.0.0.1:1234)
```

Not all sets support IP multicast, but we have tested it out on Snom,[10] Linksys/Cisco, Polycom (firmware 4.x or later), and Aastra, and it works very well.

Multicast Paging on Cisco SPA Telephones

The multicast paging feature on Cisco SPA phones is a bit strange, but once configured it works fine. The trick of it is that the address you put into the phone is not the multicast address that the page is sent across, but rather a sort of signaling channel.

What we have found is that you can make this address the same as the multicast address, but simply use a different port number.

The dialplan looks like this:

```
exten => *724,1,Page(MulticastRTP/linksys/239.0.0.1:1234/239.0.0.1:6061)
```

In the SPA phone, you need to log into the Administration interface and navigate to the *SIP* tab. At the very bottom of the page you will find the section called *Linksys Key System Parameters*. You need to set the following parameters:

- `Linksys Key System: Yes`
- `Multicast Address: 239.0.0.1:6061`

Note that the multicast address you assign to the phone is the one that comes second in the channel definition (in our example, the one using port 6061).

Note that you can write the `Page()` command in this format in an environment where there is a mix of SPA (f.k.a. Linksys, now Cisco) phones and other types of phones. The other phones will use the first address and will work the same as if you had used `basic` instead of `linksys`.

10 Very loud, and no way to adjust gain.

SIP-based paging adapters

There are many SIP-based paging speakers on the market. These devices are addressed in the dialplan in the exact same way as a SIP ATA connected to a UTI1 (in other words, to the system they're just a telephone set), but physically they are similar to external paging speakers. Since they auto-answer, there is often no need to pass them any extra information, the way you would need to with a SIP telephone set.

For smaller installations (where no more than perhaps a half-dozen speakers are required), these devices might be cost-effective because no other hardware is required. However, for anything larger than that (or for installation in a complex environment such as a warehouse or parking lot), you will get better performance at far less cost with a traditional analog paging system connected to the phone system by an analog (FXS) interface.

We haven't had any experience with these types of devices, but it is hoped that they would support multicast as standard. Keep this in mind if you are planning to use a large number of them. It's usually best to order one, test it out in a prototypical configuration, and then only commit to a quantity once you've verified that it does what you need.

Combination paging

In many organizations, there may be a need for both set-based and external paging. As an example, a manufacturing facility might want to use set-based paging for the office area but overhead paging for the plant and warehouse. From Asterisk's perspective, this is fairly simple to accomplish. When you call the `Page()` application, you simply specify the various resources you want to page, separated by the `&` character, and they will all be included in the conference that the `Page()` application creates.

Bringing it all together

At this point you should have a list of the various channel types that you want to page. Since `Page()` will nearly always want to signal more than one channel, we recommend setting a global variable in the `[globals]` section of your *extensions.conf* file that defines the list of channels to include, and then calling the `Page()` application with that string:

```
[globals]
MULTICAST=MulticastRTP/linksys/239.0.0.1:1234
;MULTICAST=MulticastRTP/linksys/239.0.0.1:1234/239.0.0.1:6061 ; if you have SPA phones

BOGEN=PJSIP/ATAforPaging  ; Assumes an ATA named [ATAforPaging]
PAGELIST=${MULTICAST}&${BOGEN} ; Variable names are arbitrary.
;...

[sets]
; ...
exten => *731,1,Page(${PAGELIST},i,120)
```

This example offers several possible configurations, depending on the hardware. While it is not strictly required to have a `PAGELIST` variable defined, we have found that this will tend to simplify the management of multiple paging resources, especially during the configuration and testing process.

Zone Paging

Zone paging is popular in places such as automobile dealerships, where the parts department, the sales department, and perhaps the used car department all require paging, but do not want to hear each other's pages.

In zone paging, the person sending the page needs to select which zone to page into. A zone paging controller such as a Bogen PCM2000 is generally used to allow signaling of the different zones: the `Page()` application signals the zone controller, the zone controller answers, and then an additional digit is sent to select to which zone the page is to be sent. Most zone controllers will allow for a page to all zones, in addition to combining zones (for example, a page to both the new- and used-car sales departments).

You could also have separate extensions in the dialplan going to separate ATAs (or groups of telephones), but this may prove more complicated and expensive than simply purchasing a paging controller that is designed to handle this. Zone paging doesn't require any significantly different technology, but it does require a little more thought and planning with respect to both the dialplan and the hardware.

And that's parking and paging. It's a ton of information to digest, but once you get the hang of it, it's quite logical.

Advanced Conferencing

The `ConfBridge()` application is an enhanced conferencing application in Asterisk that delivers high-definition audio and basic video conferencing. We previously introduced a basic working setup for `ConfBridge()`. If you've been building out your dialplan as you read along, you'll find a basic conference bridge in your *extensions.conf* file that looks something like this:

```
exten => 221,1,NoOp()
    same => n,ConfBridge(${EXTEN})
```

In a traditional Asterisk configuration, there would be a *confbridge.conf* file where we could configure parameters to apply to various scenarios. That is still possible, but it no longer makes much sense to do it that way. So, we're going to skip right over the whole configuration file, except to say that the sample file (found at *~/src/asterisk-15.<TAB>/configs/samples/confbridge.conf.sample*) now becomes an excellent reference document, but no more than that. Read on and this should start to make sense.

First up, we need to explain that there are three types of items that can be configured for a conference, namely `bridge`, `menu`, and `user`.

The `bridge` type defines the conference rooms themselves, the `menu` type defines menus that can be accessed from the conferences, and the `user` type allows different participants in the conference to have specific configuration applied to them. For example, a large conference call might have a speaker (who will do most of the talking), an administrator (to assist the speaker), and dozens of participants (who might not be allowed to speak).

Let's lay down a subroutine to get us started:

```
[subConference]
exten => _[0-9].,1,Noop(Creating conference room for ${EXTEN})
  same => n,Goto(${ARG1})
  same => n,Noop(INVALID ARGUMENT ARG1: ${ARG1})

  same => n(admin),Noop()
  same => n,Authenticate(${ARG2}) ; Could also use ,Set(CONFBRIDGE(user,pin)=${ARG2})
  same => n,Set(ConfNum=${${EXTEN} - 1])  ; Hack: Subtract 1 to get the conf number
  same => n,Set(CONFBRIDGE(bridge,record_conference)=yes)   ; Record when admin arrives
  same => n,Set(RecordingFileName=${ConfNum}-${STRFTIME(,,%Y-%m-%d %H:%m:%S)})
  same => n,Set(CONFBRIDGE(bridge,record_file)=${RecordingFileName}) ; unique name
  same => n,Set(CONFBRIDGE(user,admin)=yes)  ; Admin
  same => n,Set(CONFBRIDGE(user,marked)=yes) ; Mark this user
  same => n,Set(CONFBRIDGE(menu,7)=decrease_talking_volume) ; Decrease gain
  same => n,Set(CONFBRIDGE(menu,9)=increase_talking_volume) ; Increase gain
  same => n,Set(CONFBRIDGE(menu,4)=set_as_single_video_src)       ; Lock video on me
  same => n,Set(CONFBRIDGE(menu,5)=release_as_single_video_src)   ; Return to talker
  same => n,Set(CONFBRIDGE(menu,6)=admin_toggle_mute_participants); Mute all but admins
  same => n,Set(CONFBRIDGE(menu,2)=participant_count)             ; How many participants?
  same => n,ConfBridge(${ConfNum})
  same => n,Return()

  same => n(participant),Noop()
  same => n,Set(ConfNum=${EXTEN})
  same => n,Set(CONFBRIDGE(user,wait_marked)=yes) ; Wait for a marked user
  same => n,Set(CONFBRIDGE(user,announce_only_user)=no) ; Wait for a marked user
  same => n,Set(CONFBRIDGE(user,music_on_hold_when_empty)=yes) ; Wait for a marked user
  same => n,Set(CONFBRIDGE(menu,7)=decrease_talking_volume) ; Decrease gain
  same => n,Set(CONFBRIDGE(menu,9)=increase_talking_volume) ; Increase gain
  same => n,ConfBridge(${ConfNum})
  same => n,Return()
```

We can set `bridge`, `user`, and `menu` parameters as in the preceding example. All of the parameters you might wish to use are documented in the *~/src/asterisk-15.<TAB>/configs/samples/confbridge.conf.sample* file.

When we call the subroutine, we can pass the user as an argument. Place the following new code in your `[sets]` context after `_55512XX` and before `*724`:

```
exten => _55512XX,1,Answer()
  same => n,Playback(tt-monkeys)

; ConfBridge
```

```
exten => *600,1,GoSub(subConference,${EXTEN:1},1(participant))   ;
exten => *601,1,GoSub(subConference,${EXTEN:1},1(admin,4242))    ;

exten => *724,1,Noop(Page)
  same => n,Set(ChannelsToPage=${UserA_DeskPhone}&${UserA_SoftPhone}&${UserB_DeskPhone})
  same => n,Page(${ChannelsToPage},i,120)
```

If you dial *600, you will be joined as a participant. If you dial *601, you will be asked for the PIN (4242), and will join as an administrator. We used dialplan labels to control the call flow into the subroutine. It's easy to read, and easy to administer.

In Chapter 15 we'll explore how to use an external database to store and retrieve these parameters, rather than hardcoding them in the dialplan.

Video Conferencing

The conference engine in Asterisk can handle video, but it is a very simplified offering, and you should evaluate it carefully to ensure it meets your needs. Some of the more serious limitations include:

- All video participants must be using the same video codec; no video transcoding is available in Asterisk.
- There is no video multiplexing in Asterisk; only one video source can be shown at a time to a participant.

For a user to be able to use video (whether in a conference, or just for normal calls), they need to have the video codecs enabled. This can be done by modifying the `allow` field in the `asterisk.endpoints` table, and adding `'h264,vp8'` to the `allow` field. Make sure you don't remove the codecs that are already there (for example, the `ulaw` audio codec). A functional entry in that field might look like:

```
ulaw,h264,vp8
```

Before attempting to use video with your conferences, make sure your sets are able to use it with direct desk-to-desk calls. If you can use video conferencing between your sets, it's likely it'll also work in your conference rooms.

In Chapter 20, we'll dive into WebRTC, which is where we'll explore more powerful concepts in the delivery of multimedia communication, including conferencing.

Conclusion

In this chapter we explored the *features.conf* file, which contains the functionality for enabling DTMF-based transfers, enabling the recording of calls during a call, and configuring parking lots for one or more companies. We also looked at various ways of announcing calls and information to people in the office using a multitude of paging methods, including traditional overhead paging systems and multicast paging to

the phone sets on employees' desks. After that we delved into the `ConfBridge()` application, which is extremely flexible in configuration and rich in available features. This exploration of the various methods of implementing the traditional parking, paging, and conferencing features in a modern way hopefully shows you the flexibility Asterisk can offer.

CHAPTER 12
Automatic Call Distribution Queues

An Englishman, even if he is alone, forms an orderly queue of one.
—George Mikes

Automatic call distribution (ACD), or call queuing, provides a way for a PBX to queue up incoming calls from a group of users. It aggregates multiple calls into a holding pattern, assigns each call a rank, and determines the order in which that call should be delivered to an available agent (typically, first in first out). When an agent becomes available, the highest-ranked caller in the queue is delivered to that agent, and everyone else moves up a rank.

If you have ever called an organization and heard "all of our representatives are busy," you have experienced ACD. The advantage of ACD to the callers is that they don't have to keep dialing back in an attempt to reach someone, and the advantages to the organizations are that they are able to better serve their customers and to temporarily handle situations where there are more callers than there are agents.[1]

There are two types of call centers: inbound and outbound. ACD refers to the technology that handles inbound call centers, whereas the term *Dialer* (or *Predictive Dialer*) refers to the technology that handles outbound call centers. In this book we will primarily focus on inbound calling.

1 It is a common misconception that a queue can allow you to handle more calls. This is not strictly true: your callers will still want to speak to a live person, and they will only be willing to wait for so long. In other words, if you are short-staffed, your queue could end up being nothing more than an obstacle to your callers. This is the same whether you're on the phone or at the Walmart checkout. Nobody likes to wait in line. The ideal queue is invisible to the callers, since their calls get answered immediately without them having to wait.

We've all been frustrated by poorly designed and managed queues: enduring hold music from a radio that isn't in tune, mind-numbing wait times, and pointless messages that tell you every 20 seconds how important your call is, despite that fact that you've been waiting for 30 minutes and have heard the message so many times you can quote it from memory. From a customer service perspective, queue design may be one of the most important aspects of your telephone system. As with an automated attendant, what must be kept in mind above all else is that *your callers are not interested in holding in a queue*. They called because *they want to talk to you*. All your design decisions must keep this crucial fact front-and-center in your mind: people want to talk to other people, not to your phone system.[2]

The purpose of this chapter is to teach you how to create and design queues that get callers to their intended destinations as quickly and painlessly as possible.

In this chapter, we may flip back and forth between the usage of the terms *queue members* and *agents*. Since we're not going to spend much time on the Asterisk module named `chan_agent` (using `AgentLogin()`), we need to make it clear that in this book, when we use the term *agent*, we're referring to an endpoint—a human being, and not the channel technology in Asterisk named `chan_agent`. Read on, and this should make more sense.

Creating a Simple ACD Queue

To start with, we're going to create a simple ACD queue. It will accept callers and attempt to deliver them to a member of the queue.

In Asterisk, the term *member* refers to a channel (typically a SIP peer) assigned to a queue that can be dialed, such as `SIP/0000FFFF0001`. An *agent* technically refers to the Agent channel also used for dialing endpoints. Unfortunately, the Agent channel is a deprecated technology in Asterisk, as it is limited in flexibility and can cause unexpected issues that can be hard to diagnose and resolve. We will not be covering the use of `chan_agent`, so be aware that we will generally use the term *member* to refer to the telephone device and *agent* to refer to the person who handles the call. Since one isn't generally effective without the other, either term may refer to both.

2 There are several books available that discuss call center metrics and available queuing strategies, such as James C. Abbott's *The Executive Guide to Call Center Metrics* (Robert Houston Smith).

We'll create the queue(s) in the *queues.conf* file, and manually add queue members to it through the Asterisk console. In the section "Queue Members" on page 215, we'll look into how to create a dialplan that allows us to dynamically add and remove queue members (as well as pause and unpause them).

The first step is to create an empty *agents.conf* file in your */etc/asterisk* configuration directory. We will not use or edit this file, but the `app_queue` module expects to find it, and will not load if it does not exist:

```
$ cd /etc/asterisk

$ sudo -u asterisk touch agents.conf
```

Since we haven't done so yet, we're also going to configure basic music on hold, using the sample file:

```
$ sudo cp ~/src/asterisk-16.<TAB>/configs/samples/musiconhold.conf.sample \
/etc/asterisk/musiconhold.conf

$ sudo chown asterisk:asterisk /etc/asterisk/musiconhold.conf
```

Next you need to create the *queues.conf* file, which we're not going to edit because we'll be creating our queues in the database (it just needs to be there):

```
$ sudo touch -u asterisk queues.conf
```

Next, we're going to create some queues in our database:

```
MySQL> INSERT INTO `asterisk`.`queues`
(name,strategy,joinempty,leavewhenempty,ringinuse,autofill,musiconhold, \
monitor_format,monitor_type)

VALUES
'sales','rrmemory','unavailable,invalid,unknown','unavailable,invalid,unknown','no','yes',\
'default','wav','MixMonitor'),
('support','rrmemory','unavailable,invalid,unknown','unavailable,invalid,unknown','no',\
'yes','default','wav','MixMonitor') ;
```

This will give us two queues named `sales` and `support`. You can name them anything you want, but we will be using these names later in the book, so if you use different queue names from what we've recommended here, make note of your choices for future reference.

We have also defined the parameters outlined in Table 12-1.

Table 12-1. Sample queue parameters

Parameter	Purpose
`strategy=rrmemory`	Use the round robin with memory strategy
`joinempty=unavailable,invalid,unknown`	Do not join the queue when no members available
`leavewhenempty=unavailable,inva lid,unknown`	Leave the queue when no members available

Parameter	Purpose
`ringinuse=no`	Don't ring members when already `InUse` (prevents multiple calls to an agent)
`autofill=yes`	Distribute all waiting callers to available members
`musiconhold=default`	Play music from the `[default]` class (see `musiconhold.conf`)

The `strategy` we'll employ is `rrmemory`, which stands for round robin with memory. The `rrmemory` strategy works by rotating through the agents in the queue in sequential order, keeping track of which agent got the last call, and presenting the next call to the next agent. When it gets to the last agent, it goes back to the top (as agents log in, they are added to the end of the list).

A Few Notes on Strategies

`ringall`
: Rings all available members (default). This distribution strategy doesn't really count as ACD. In traditional telephony terms, this would be known as a *ring group*.

`leastrecent`
: Rings the interface that least recently received a call. In a queue where there are many calls of roughly the same duration, this can work. It doesn't work as well if an agent has been on a call for an hour, and their colleagues all got their last call 30 minutes ago, because the agent who just finished the 60-minute call will get the next one.

`fewestcalls`
: Rings the interface that has completed the fewest calls in this queue. This can be unfair if calls are not always of the same duration. An agent could handle three calls of 15 minutes each and her colleague had four 5-second calls; the agent who handled three calls will get the next one.

`random`
: Rings a random interface. This actually can work very well and end up being very fair in terms of evenly distributing calls among agents.

`rrmemory`
: Rings members in a round-robin fashion, remembering where it left off last for the next caller. This can also work out to be very fair, but not as much as `random`.

`linear`
: Rings members in the order specified, always starting at the beginning of the list. This works if you have a team where there are some agents who are supposed to

handle most calls, and other agents who should only get calls if the primary agents are busy.

`wrandom`
: Rings a random member, but uses the members' penalties as a weight. Worth considering in a larger queue with complex weighting among the agents.

We've set `joinempty` to `no` since it is generally bad form to put callers into a queue where there are no agents available to take their calls.

You could set this to `yes` for ease of testing, but we would not recommend putting it into production unless you are using the queue for some function that is not about getting your callers to your agents. Nobody wants to wait in a line that is not going anywhere.

The `leavewhenempty` option is used to control whether callers should fall out of the `Queue()` application and continue on in the dialplan if no members are available to take their calls. We've set this to `yes` because you won't normally want callers waiting in a queue with no logged-in agents.

From a business perspective, you should be telling your agents to clear all calls out of the queue before logging off for the day. If you find that there are a lot of calls queued up at the end of the day, you might want to consider extending someone's shift to deal with them. Otherwise, they'll just add to your stress when they call back the next day, in a worse mood.

You can use `GotoIfTime()` near the end of the day to redirect callers to voicemail, or some other appropriate location in your dialplan, while your agents clear out any remaining calls in the queue.

We'll want `ringinuse` to be `no`, which tells Asterisk not to ring members when their devices are already ringing. The purpose of setting `ringinuse` to `no` is to avoid multiple calls to the same member from one or more queues.

It should be mentioned that `joinempty` and `leavewhenempty` are looking for either no members logged into the queue, or all members unavailable. Agents that are `Ringing` or `InUse` are not considered unavailable, so will not block callers from joining the queue or cause them to be kicked out when `joinempty=no` and/or `leavewhenempty=yes`.

The `autofill` option tells the queue to distribute all waiting callers to all available members immediately. Previous versions of Asterisk would only distribute one caller at a time, which meant that while Asterisk was signaling an agent, all other calls were held (even if other agents were available) until the first caller in line had been connected to an agent (which obviously led to bottlenecks in older versions of Asterisk where large, busy queues were being used). Unless you have a particular need for backward compatibility, *this option should always be set to* `yes`.

Verify that your */etc/asterisk/extconfig* file contains the following lines:

```
queues => odbc,asterisk,queues
queue_members => odbc,asterisk,queue_members
```

Save and reload your queue configuration from the Asterisk CLI:

```
*CLI> queues reload
```

Verify that your queues were loaded into memory (don't forget to ensure an empty *agents.conf* file exists):

```
localhost*CLI> queue show
support      has 0 calls (max unlimited) in 'rrmemory' strategy
(0s holdtime, 0s talktime), W:0, C:0, A:0, SL:0.0% within 0s
   No Members
   No Callers

sales        has 0 calls (max unlimited) in 'rrmemory' strategy
(0s holdtime, 0s talktime), W:0, C:0, A:0, SL:0.0% within 0s
   No Members
   No Callers
```

The output of `queue show` provides various pieces of information, including those parts detailed in Table 12-2.

Table 12-2. Output of queue show CLI command

Field	Description
W:	Queue weight
C:	Number of calls presented to this queue
A:	Number of calls that have been answered by a member
SL:	Service level

Now that you've created the queues, you need to configure your dialplan to allow calls to enter the queue.

Add the following dialplan logic to the *extensions.conf* file (somewhere in the `[sets]` context):

```
exten => 610,1,Noop()
  same => n,Progress()
  same => n,Queue(sales)
  same => n,Hangup()
```

```
exten => 611,1,Noop()
  same => n,Progress()
  same => n,Queue(support)
  same => n,Hangup()
```

Save the changes to your *extensions.conf* file, and reload the dialplan with the `dialplan reload` CLI command.

If you dial extension 610 or 611 at this point, you will end up with output like the following:

```
    == Setting global variable 'SIPDOMAIN' to '172.29.1.178'
    -- Executing [610@sets:1] NoOp("PJSIP/SOFTPHONE_A-00000004", "") in new stack
    -- Executing [610@sets:2] Progress("PJSIP/SOFTPHONE_A-00000004", "") in new stack
    -- Executing [610@sets:3] Queue("PJSIP/SOFTPHONE_A-00000004", "test") in new stack
       > 0x7facc801ed60 -- Strict RTP learning after remote  set to: 172.29.1.166:4022
    -- Started music on hold, class 'testmoh', on channel 'PJSIP/SOFTPHONE_A-00000004'
       > 0x7facc801ed60 -- Strict RTP switching to RTP target 172.29.1.166:4022 as source
       > 0x7facc801ed60 -- Strict RTP learning complete - Locking on 172.29.1.166:4022
    -- Stopped music on hold on PJSIP/SOFTPHONE_A-00000004
  == Spawn extension (sets, 610, 3) exited non-zero on 'PJSIP/SOFTPHONE_A-00000004'
```

Note that *you won't join the queue at this point* because there are no agents in the queue to answer calls. We have `joinempty=no` and `leavewhenempty=yes` configured, so callers will not be placed into the queue. (This would be a good opportunity to experiment with the `joinempty` and `leavewhenempty` options in *queues.conf* to better understand their impact on queues.)

In the next section, we'll demonstrate how to add members to your queue (as well as other member interactions with the queue, such as pause/unpause).

Queue Members

Queues aren't very useful without someone to answer the calls that come into them, so we need a method for allowing agents to be logged into the queues to answer calls. There are various ways of going about this, so we'll show you how to add members to the queue both manually (as an administrator, via either the CLI or hardcoded in the `queue_members` table) and dynamically (as the agent, through an extension defined in the dialplan). We'll start with the Asterisk CLI method, which allows you to easily add members to the queue for testing, with minimal dialplan changes. Next we'll show how you can define members in the `queue_members` table. Finally, we'll show you how to add dialplan logic that allows agents to log themselves into and out of the queues and to pause and unpause themselves in queues they are logged into (this is likely the best method for production).

Controlling Queue Members via the CLI

We can add queue members to any available queue through the Asterisk CLI command `queue add`. The format of the `queue add` command is (all on one line):

```
*CLI> queue add member channel to queue [[[penalty penalty] as
membername] state_interface interface]
```

The *`channel`* is the channel we want to add to the queue, such as `SIP/0000FFFF0003`, and the *`queue`* name will be something like `support` or `sales`—any queue name that exists in */etc/asterisk/queues.conf*. For now we'll ignore the *`penalty`* option, but we'll discuss it in "Advanced Queues" on page 223 (`penalty` is used to control the rank of a member within a queue, which can be important for agents who are logged into multiple queues, or have differing skills). We can define the *`membername`* to provide details to the queue-logging engine.

The `state_interface` option informs the queue of the device state to be monitored for this agent. The details of how to work with device states are discussed in Chapter 13. Go ahead and work through that chapter, and then come back here and continue on. Don't worry, we'll wait.

Now that you've added `callcounter=yes` to *sip.conf* (we'll be using SIP channels throughout the rest of our examples), let's see how to add members to our queues from the Asterisk CLI.

Adding a queue member to the `support` queue can be done with the `queue add member` command:

```
*CLI> queue add member PJSIP/SOFTPHONE_B to support

Added interface 'PJSIP/SOFTPHONE_B' to queue 'support'
```

A query of the queue will verify that our new member has been added:

```
*CLI> queue show support

support has 0 calls (max unlimited) in 'rrmemory' strategy (0s holdtime, 0s talktime),
W:0, C:0, A:0, SL:0.0%, SL2:0.0% within 0s
   Members:
      PJSIP/SOFTPHONE_B (ringinuse disabled) (dynamic) (Not in use) has taken no calls yet
   No Callers
```

To remove a queue member, you would use the `queue remove member` command:

```
*CLI> queue remove member PJSIP/SOFTPHONE_B from support

Removed interface PJSIP/SOFTPHONE_B from queue 'support'
```

Of course, you can use the `queue show` command again to verify that your member has been removed from the queue:

```
*CLI> queue show support
```

```
support has 0 calls (max unlimited) in 'rrmemory' strategy (0s holdtime, 0s talktime),
W:0, C:0, A:0, SL:0.0%, SL2:0.0% within 0s
   Members:
      PJSIP/SOFTPHONE_B (ringinuse disabled) (dynamic) (Not in use) has taken no calls yet
   No Callers
```

We can also pause and unpause members in a queue from the Asterisk console, with the `queue pause member` and `queue unpause member` commands. They take a similar format to the previous commands we've been using:

```
*CLI> queue pause member PJSIP/SOFTPHONE_B queue support reason Callbacks

paused interface 'PJSIP/SOFTPHONE_B' in queue 'support' for reason 'Callbacks'

*CLI> queue show support
support      has 0 calls (max unlimited) in 'rrmemory' strategy
(0s holdtime, 0s talktime), W:0, C:0, A:0, SL:0.0% within 0s
   Members:
      SIP/0000FFFF0001 (dynamic) (paused) (Not in use) has taken no calls yet
   No Callers

*CLI> queue show support

support has 0 calls (max unlimited) in 'rrmemory' strategy (0s holdtime, 0s talktime),
 W:0, C:0, A:0, SL:0.0%, SL2:0.0% within 0s
   Members:
      PJSIP/SOFTPHONE_B (ringinuse disabled) (dynamic) (paused:Callbacks) (Not in use)
has taken no calls yet
   No Callers
```

By adding a reason for pausing the queue member, such as `lunchtime`, you ensure that your queue logs will contain some additional information that may be useful. Here's how to unpause the member:

```
*CLI> queue unpause member PJSIP/SOFTPHONE_B queue support reason FinishedCallBacks

unpaused interface 'PJSIP/SOFTPHONE_B' in queue 'support' for reason 'FinishedCallbacks'
```

In a production environment, the CLI would not normally be the best way to control the state of agents in a queue. Instead, there are dialplan applications that allow agents to inform the queue as to their availability.

Defining Queue Members in the queue_members Table

If you define a queue member in the `asterisk.queue_members` table of the database, that member will always be logged into the queue. This usually doesn't work well if your members are human beings, since humans tend to get up and move about.

Within each queue definition, you simply define the members thus:

```
MySQL> insert into `asterisk`.`queue_members`
(queue_name,interface,penalty)

VALUES
'hotline','PJSIP/SOME_NON_HUMAN','0');
```

In a typical queue (one in which you have a group of people responsible for answering calls), you will find that defining the members in the `queue_members` table might not serve you well. Human agents usually need to be able to log in and out (and not be automatically logged in whenever the queue is reloaded). We do not recommend defining members in the `queue_members` table unless they have some other purpose (such as a bank of devices that answer calls, where you want to use the queue to load-balance calls to the device pool, or a ring group, where all phones ring for all calls all the time regardless of whether anyone is sitting near the phone).

Controlling Queue Members with Dialplan Logic

In a call center staffed by live agents, it is most common to have the agents themselves log in and log out at the start and end of their shifts (or whenever they go for lunch, or to the bathroom, or are otherwise not available to the queue).

To enable this, we will make use of the following dialplan applications:

- `AddQueueMember()`
- `RemoveQueueMember()`

While logged into a queue, it may be that an agent needs to put themself into a state where they are temporarily unavailable to take calls. The following applications will allow this:

- `PauseQueueMember()`
- `UnpauseQueueMember()`

The `Add/Remove` applications are used to log in and log out, and `Pause/Unpause` are used for short periods of agent unavailability. The difference is simply that `Pause` and `Unpause` set the member as unavailable/available without actually removing them from the queue. This is mostly useful for reporting purposes (if a member is paused, the queue supervisor can see that they are logged into the queue, but simply not available to take calls at that moment). If you're not sure which one to use, we recommend that the agents use `Add/Remove` whenever they are not physically at their phone, and `Pause/Unpause` when they are at their desk, but temporarily not available.

If in doubt, it's usually better to have your agents log out.

Using Pause and Unpause

In some environments, `Pause` and `Unpause` are used for all activities during the day that render an agent unavailable (such as during the lunch hour and when performing work that is not queue-related). In most call centers, however, if an agent is not

beside the phone and ready to take a call at that moment, they should not be logged in at all, even if they are only going to be away from their desk for a few minutes (such as for a bathroom break).

Some supervisors like to use the `Pause/Unpause` settings as a sort of punch clock, so that they can track when their staff arrive for work and leave at the end of the day, and how long they spend at their desks and on breaks. This may not be a sound practice, since the purpose of these applications is to inform the queue as to agent availability, with activity tracking a secondary function.

An important thing to note here relates to the `joinempty` setting in the `asterisk.queues` table, which was discussed earlier. If an agent is paused, they are still logged into the queue. Let's say it is near the end of the day, and one agent put themself into pause a few hours earlier to work on a project. All the other agents have logged out and gone home. A call comes in. The queue will note that an agent is logged into the queue, and will therefore queue the call, even though the reality is that there are no people actually staffing that queue at that time. This caller may end up holding in an unstaffed queue indefinitely.

In short, agents who are not sitting at their desks and planning to be available to take calls in the next few minutes should log out. `Pause/Unpause` should only be used for brief moments of unavailability (if at all). If you want to use your phone system as a punch clock, there are lots of great ways to do that using Asterisk, but the queue member applications are not the way we would recommend.

Let's build some simple dialplan logic that will allow our agents to indicate their availability to the queue. We are going to use the `CUT()` dialplan function to extract the name of our channel from our call to the system, so that the queue will know which channel to log into the queue.

We have built this dialplan to show a simple process for logging into and out of a queue, and changing the paused status of a member in a queue. We are doing this only for a single queue that we previously defined in the *queues.conf* file. The status channel variables that the `AddQueueMember()`, `RemoveQueueMember()`, `PauseQueueMember()`, and `UnpauseQueueMember()` applications set might be used to `Playback()` announcements to the queue members after they've performed certain functions to let them know whether they have successfully logged in/out or paused/unpaused:

```
exten => *731,1,Page(${PAGELIST},i,120)

exten => *732,1,Verbose(2,Logging In Queue Member)
   same => n,Set(MemberChannel=${CHANNEL(channeltype)}/${CHANNEL(endpoint)})
   same => n,AddQueueMember(support,${MemberChannel})
   same => n,Verbose(1,${AQMSTATUS}) ; ADDED, MEMBERALREADY, NOSUCHQUEUE
   same => n,Playback(agent-loginok)
   same => n,Hangup()

exten => *733,1,Verbose(2,Logging Out Queue Member)
```

```
   same => n,Set(MemberChannel=${CHANNEL(channeltype)}/${CHANNEL(endpoint)})
   same => n,RemoveQueueMember(support,${MemberChannel})
   same => n,Verbose(1,${RQMSTATUS}) ; REMOVED, NOTINQUEUE, NOSUCHQUEUE
   same => n,Playback(agent-loggedoff)
   same => n,Hangup()

exten => *734,1,Verbose(2,Pause Queue Member)
   same => n,Set(MemberChannel=${CHANNEL(channeltype)}/${CHANNEL(endpoint)})
   same => n,PauseQueueMember(support,${MemberChannel})
   same => n,Verbose(1,${PQMSTATUS}) ; PAUSED, NOTFOUND
   same => n,Playback(dictate/paused)
   same => n,Hangup()

exten => *735,1,Verbose(2,Unpause Queue Member)
   same => n,Set(MemberChannel=${CHANNEL(channeltype)}/${CHANNEL(endpoint)})
   same => n,UnpauseQueueMember(support,${MemberChannel})
   same => n,Verbose(1,${UPQMSTATUS}) ; UNPAUSED, NOTFOUND
   same => n,Playback(agent-loginok)
   same => n,Hangup()

exten => *98,1,NoOp(Access voicemail retrieval.)
```

Automatically Logging Into and Out of Multiple Queues

It is quite common for an agent to be a member of more than one queue. Rather than having a separate extension for logging into each queue (or demanding information from the agents about which queues they want to log into), this code uses the Asterisk database (`astdb`) to store queue membership information for each agent, and then loops through each queue the agents are a member of, logging them into each one in turn.

In order for this code to work, an entry similar to the following will need to be added to the AstDB via the Asterisk CLI. For example, the following would store the member `SOFTPHONE_A` as being in both the `support` and `sales` queues:[3]

```
*CLI> database put queue_agent SOFTPHONE_A/available_queues support^sales
```

You will need to do this once for each agent, regardless of how many queues they are members of.

If you then query the Asterisk database, you should get a result similar to the following:

```
pbx*CLI> database show queue_agent
/queue_agent/SOFTPHONE_A/available_queues         : support^sales
```

The following dialplan code is an example of how to allow this queue member to be automatically added to both the `support` and `sales` queues. We've defined a

3 We're going to use the ^ character as a delimiter. You could probably use another character instead, just so long as it's not one the Asterisk parser would see as a normal delimiter (and thus get confused by). So avoid commas, semicolons, and so forth.

subroutine that is used to set up three channel variables (`MemberChannel`, `MemberChanType`, `AvailableQueues`). These channel variables are then used by the login (`*736`), logout (`*737`), pause (`*738`), and unpause (`*739`) extensions. Each of the extensions uses the `subSetupAvailableQueues` subroutine to set these channel variables and to verify that the AstDB contains a list of one or more queues for the device the queue member is calling from.

Near the end of your *extensions.conf* file, where you've put your subroutines, add the following:

```
[subSetupAvailableQueues]
; This subroutine is used by the various login/logout/pausing/unpausing routines
; in our multiple queue login example.
;
exten => start,1,Verbose(2,Checking for available queues)
; Get the current channel's peer name
   same => n,Set(MemberChannel=${CHANNEL(endpoint)})
; Get the current channel's technology type
   same => n,Set(MemberChanType=${CHANNEL(channeltype)})
; Get the list of queues available for this agent
   same => n,Set(AvailableQueues=${DB(queue_agent/${MemberChannel}/available_queues)})
; if there are no queues assigned to this agent we'll handle it in the
; no_queues_available extension
   same => n,GotoIf($[${ISNULL(${AvailableQueues})}]?no_queues_available,1)
   same => n,Return()

exten => no_queues_available,1,Verbose(2,No queues available for agent ${MemberChannel})
; playback a message stating the channel has not yet been assigned
   same => n,Playback(silence/1&channel&not-yet-assigned)
   same => n,Hangup()
```

Next, in your `[sets]` context, add the following:

```
; Logging into multiple queues via the AstDB system
exten => *736,1,Verbose(2,Logging into multiple queues per the database values)
; get the available queues for this channel
   same => n,GoSub(subSetupAvailableQueues,start,1())
   same => n,Set(QueueCounter=1)  ; setup a counter variable
; using CUT(), get the first listed queue returned from the AstDB
; Note that we've used '^' as our delimiter
   same => n,Set(WorkingQueue=${CUT(AvailableQueues,^,${QueueCounter})})
; While the WorkingQueue channel variable contains a value, loop
   same => n,While($[${EXISTS(${WorkingQueue})}])
; AddQueueMember(queuename[,interface[,penalty[,options[,membername
;  [,stateinterface]]]]])
; Add the channel to a queue, setting the interface for calling
; and the interface for monitoring of device state
; *** This should all be on a single line
     same => n,AddQueueMember(
     ${WorkingQueue},${MemberChanType}/${MemberChannel},,,${MemberChanType}/${MemberChannel})
     same => n,Set(QueueCounter=$[${QueueCounter} + 1])    ; increase our counter
; get the next available queue; if it is null our loop will end
     same => n,Set(WorkingQueue=${CUT(AvailableQueues,^,${QueueCounter})})
     same => n,EndWhile()
; let the agent know they were logged in okay
   same => n,Playback(silence/1&agent-loginok)
   same => n,Hangup()
```

```
exten => no_queues_available,1,Verbose(2,No queues available for ${MemberChannel})
   same => n,Playback(silence/1&channel&not-yet-assigned)
   same => n,Hangup()

; Used for logging agents out of all configured queues per the AstDB
exten => *737,1,Verbose(2,Logging out of multiple queues)
; Because we reused some code, we've placed the duplicate code into a subroutine
   same => n,GoSub(subSetupAvailableQueues,start,1())
   same => n,Set(QueueCounter=1)
   same => n,Set(WorkingQueue=${CUT(AvailableQueues,^,${QueueCounter})})
   same => n,While($[${EXISTS(${WorkingQueue})}])
   same => n,RemoveQueueMember(${WorkingQueue},${MemberChanType}/${MemberChannel})
   same => n,Set(QueueCounter=$[${QueueCounter} + 1])
   same => n,Set(WorkingQueue=${CUT(AvailableQueues,^,${QueueCounter})})
   same => n,EndWhile()
   same => n,Playback(silence/1&agent-loggedoff)
   same => n,Hangup()

; Used for pausing agents in all available queues
exten => *738,1,Verbose(2,Pausing member in all queues)
   same => n,GoSub(subSetupAvailableQueues,start,1())
   ; if we don't define a queue, the member is paused in all queues
   same => n,PauseQueueMember(,${MemberChanType}/${MemberChannel})
   same => n,GotoIf($[${PQMSTATUS} = PAUSED]?agent_paused,1:agent_not_found,1)

exten => agent_paused,1,Verbose(2,Agent paused successfully)
   same => n,Playback(dictate/paused)
   same => n,Hangup()

; Used for unpausing agents in all available queues
exten => *739,1,Verbose(2,UnPausing member in all queues)
   same => n,GoSub(subSetupAvailableQueues,start,1())
   ; if we don't define a queue, then the member is unpaused from all queues
   same => n,UnPauseQueueMember(,${MemberChanType}/${MemberChannel})
   same => n,GotoIf($[${UPQMSTATUS} = UNPAUSED]?agent_unpaused,1:agent_not_found,1)

exten => agent_unpaused,1,Verbose(2,Agent paused successfully)
   same => n,Playback(silence/1&available)

; Used by both pausing and unpausing dialplan functionality
exten => agent_not_found,1,Verbose(2,Agent was not found)
   same => n,Playback(silence/1&cannot-complete-as-dialed)
```

You could further refine these login and logout routines to take into account that the `AQMSTATUS` and `RQMSTATUS` channel variables are set each time `AddQueueMember()` and `RemoveQueueMember()` are used. For example, you could set a flag that lets the queue member know they have not been added to a queue, or even add recordings or text-to-speech systems to play back the particular queue that is producing the problem. Or, if you're monitoring this via the Asterisk Manager Interface, you could have a screen pop, or use `JabberSend()` to inform the queue member via instant messaging, or…(ain't Asterisk fun?).

Advanced Queues

In this section we'll take a look at some of the finer-grained queue controls, such as options for controlling announcements and when callers should be placed into (or removed from) the queue. We'll also look at penalties and priorities, exploring how we can control the agents in our queue by giving preference to a pool of agents, and then increasing that pool dynamically based on the wait times in the queue. Finally, we'll look at using local channels as queue members, which gives us the ability to perform dialplan tricks prior to connecting the caller to an agent.

Priority Queue (Queue Weighting)

Sometimes you need to add people to a queue at a higher priority than that given to other callers. Perhaps the caller has already spent time waiting in a queue, and an agent has taken some information but realized the caller needed to be transferred to another queue. In this case, to minimize the caller's overall wait time, it might be desirable to transfer the call to a priority queue that has a higher weight (and thus a higher preference), so it will be answered quickly.

Setting a higher priority on a queue is done with the `weight` option. If you have two queues with differing weights (e.g., `support` and `support-priority`), agents assigned to both queues will be passed calls from the higher-priority queue in preference to calls from the lower-priority queue. Those agents will not take any calls from the lower-priority queue until the higher-priority queue is cleared. (Normally, there will be some agents who are assigned only to the lower-priority queue, to ensure that those calls are dealt with in a timely manner.) For example, if we place queue member James Shaw into both the `support` and `support-priority` queues, callers in the `support-priority` queue will have a preferred standing with James over callers in the `support` queue.

Let's take a look at how we could make this work. First, we need to create a new queue that's similar to the support queue except for the `weight` option.

```
MySQL> INSERT INTO `asterisk`.`queues`
(name,strategy,joinempty,leavewhenempty,ringinuse,autofill,musiconhold,monitor_format,
monitor_type,weight)

VALUES
('support-priority','rrmemory','unavailable,invalid,unknown','unavailable,invalid,unknown',
'no','yes','default','wav','MixMonitor','10');
```

With our new queue configured, we can now create two extensions to transfer callers to. This can be done wherever you would normally place your dialplan logic to perform transfers. We're going to use the `LocalSets` context, which we've previously enabled as the starting context for our devices:

```
exten => 611,1,Noop()
  same => n,Progress()
  same => n,Queue(support)
  same => n,Hangup()

exten => 612,1,Noop()
  same => n,Progress()
  same => n,Queue(support-priority)
  same => n,Hangup()

exten => *724,1,Noop(Page)
```

The only other configuration left to do is to make sure some or all of your queue members are placed in both queues.

Queue Member Priority

Within a queue, we can apply a penalty to members in order to lower their preference for being called when there are people waiting in a particular queue. For example, we may penalize queue members when we want them to be a member of a queue, but only receive calls when the queue gets full enough that all our preferred agents are unavailable. By defining different penalties for each member of the queue,[4] we can help control the preference for where callers are delivered, but still ensure that other queue members will be available to answer calls if the preferred member is unavailable.

Penalties can also be defined using `AddQueueMember()`. We'll modify our multiple queue login to provide the required penalties.

First, let's update our AstDB to include penalties for a member:

```
*CLI> database put queue_agent SOFTPHONE_A/penalty 0^2

*CLI> database show queue agent

/queue_agent/SOFTPHONE_A/available_queues         : support^sales
/queue_agent/SOFTPHONE_A/penalty                  : 0^2
```

Next, a few tweaks to our dialplan.

The subroutine needs a new line (some code has been removed for brevity, replaced with `;...`):

```
[subSetupAvailableQueues]
; ...
; Get the list of queues available for this agent
   same => n,Set(AvailableQueues=${DB(queue_agent/${MemberChannel}/available_queues)})
   same => n,Set(MemberPenalties=${DB(queue_agent/${MemberChannel}/penalty)})
; if there are no queues assigned ...
```

4 Similar to adding ballast to a jockey or racing car.

The [sets] context requires a couple of new lines as well (some code has been removed for brevity, replaced with ;...). Only insert/change the code written in bold.

```
exten => *736,1,Verbose(2,Logging into multiple queues per the database values)
; ...
   same => n,Set(WorkingQueue=${CUT(AvailableQueues,^,${QueueCounter})})
   same => n,Set(WorkingPenalty=${CUT(MemberPenalties,^,${QueueCounter})})
; While the WorkingQueue ...
; ...
     same => n,Set(WorkingQueue=${CUT(AvailableQueues,^,${QueueCounter})})
     same => n,Set(WorkingPenalty=${CUT(MemberPenalties,^,${QueueCounter})})
     same => n,EndWhile()
; ...
```

These examples are probably not suitable for a production environment (we'd use purpose-built MySQL tables for this sort of thing rather than AstDB), but it gives you an idea of how the dialplan can be used to apply dynamic logic to more complex configuration scenarios.

Changing Penalties Dynamically (queuerules)

Using the asterisk.queuerules table, it is possible to define rules that change the values of the QUEUE_MIN_PENALTY and QUEUE_MAX_PENALTY channel variables. The QUEUE_MIN_PENALTY and QUEUE_MAX_PENALTY channel variables are used to control which members of a queue are preferred for servicing callers. Let's say we have a queue called support, and we have five queue members with various penalties ranging from 1 through 5. If, prior to a caller entering the queue, the QUEUE_MIN_PENALTY channel variable is set to a value of 2 and the QUEUE_MAX_PENALTY is set to a value of 4, only queue members whose penalties are set to values ranging from 2 through 4 will be considered available to answer that call:

```
same => n,Set(QUEUE_MIN_PENALTY=2)   ; set minimum member penalty
same => n,Set(QUEUE_MAX_PENALTY=4)   ; set maximum member penalty
same => n,Queue(support)             ; entering the queue with min and max
                                     ; member penalties to be used
```

What's more, during the caller's stay in the queue, we can dynamically change the values of QUEUE_MIN_PENALTY and QUEUE_MAX_PENALTY for that caller. This allows either more or a different set of queue members to be used, depending on how long the caller waits in the queue. For instance, in the previous example, we could modify the minimum penalty to 1 and the maximum penalty to 5 if the caller has to wait more than 60 seconds in the queue.

The sample file *~/src/asterisk-15.<TAB>/configs/samples/queuerules.conf.sample* contains an excellent reference for how queue rules work.

The rules are defined using the `asterisk.queuerules` table. Multiple rules can be created in order to facilitate different penalty changes throughout the call. Let's take a look at how we might choose to define a rule:

```
MySQL> insert into `asterisk`.`queue_rules`
(rule_name,time,min_penalty,max_penalty)

VALUES
('more_members',60,5,1);
```

New rules will affect only new callers entering the queue, not existing callers already holding.

We've named the rule `more_members` and defined the following values:

`60`
: The number of seconds to wait before changing the penalty values.

`5`
: The new `QUEUE_MAX_PENALTY`.

`1`
: The new `QUEUE_MIN_PENALTY`.

We can now tell our queues to make use of it.

```
MySQL> update `asterisk`.`queues`

set defaultrule='more_members' where `name` in ('sales','support')
```

The *queuerules.conf.sample* file shows that these rules are quite flexible. If you want fine-grained control over call prioritization, some additional lab work may be worth your while.

Announcement Control

Asterisk has the ability to play several announcements to callers waiting in the queue. For example, you might want to announce the caller's position in the queue, announce the average wait time, or periodically thank your callers for waiting (or whatever your audio files say). It's important to carefully tune the values that control when these announcements are played to the callers, because announcing their position, thanking them for waiting, and informing them of the average hold time too frequently is going to tend to annoy them, which is not the goal of these things.

Playing Announcements Between Music on Hold Files

Instead of handling the intricacies of announcements for each of your queues, you could alternatively (or in conjunction) utilize the announcement functionality defined in *musiconhold.conf*. Prior to playing a file for music, the announcement file will be played, and then played again between audio files. Let's say you have a 5-minute loop of audio, but you want to play a "Thank you for waiting" message every 30 seconds. You could split the audio file into 30-second segments, set their filenames as starting with `00-`, `01-`, `02-`, and so on (to keep them playing in order), and then define the `announcement`. The *musiconhold.conf* class might look something like this:

```
[moh_jazz_queue]
mode=files
sort=alpha
announcement=queue-thankyou
directory=moh_jazz_queue
```

There are several options in the `queues` table that you can use to fine-tune what and when announcements are played to your callers. The full list of queue options is available in the *~/src/asterisk-15.<TAB>/configs/samples/queues.conf.sample* file. Table 12-3 reviews a few of the more useful ones.

Table 12-3. Options related to prompt control timing within a queue

Option	Available values	Description
`announce-frequency`	Value in seconds	Defines how often we should announce the caller's position and/or estimated hold time in the queue. Set this value to zero to disable.
`min-announce-frequency`	Value in seconds	Indicates the minimum amount of time that must pass before we announce the caller's position in the queue again. This is used when the caller's position may change frequently, to prevent the caller hearing multiple updates in a short period of time.
`periodic-announce-frequency`	Value in seconds	Specifies how often to make periodic announcements to the caller.
`random-periodic-announce`	`yes`, `no`	If set to `yes`, will play the defined periodic announcements in a random order. See `periodic-announce`.
`relative-periodic-announce`	`yes`, `no`	If set to `yes`, the `periodic-announce-frequency` timer will start when the end of the file being played back is reached, instead of from the beginning. Defaults to `no`.
`announce-holdtime`	`yes`, `no`, `once`	Defines whether the estimated hold time should be played along with the periodic announcements. Can be set to `yes`, `no`, or only `once`.

Option	Available values	Description
announce-position	yes, no, limit, more	Defines whether the caller's position in the queue should be announced to them. If set to no, the position will never be announced. If set to yes, the caller's position will always be announced. If the value is set to limit, the caller will hear their position in the queue only if it is within the limit defined by announce-position-limit. If the value is set to more, the caller will hear their position only if it is beyond the number defined by announce-position-limit.
announce-position-limit	Number of zero or greater	Used if you've defined announce-position as either limit or more.
announce-round-seconds	Value in seconds	If this value is nonzero, the number of seconds is announced as well, and rounded to the value defined.

Table 12-4 defines the files that will be used when announcements are played to the caller.

Table 12-4. Options for controlling the playback of prompts within a queue

Option	Available values	Description
musicclass	Music class as defined by *musiconhold.conf*	Sets the music class to be used by a particular queue. You can also override this value with the CHANNEL(musicclass) channel variable.
queue-thankyou	Filename of prompt to play	If not defined, plays the default value ("Thank you for your patience"). If set to an empty value, prompt will not be played at all.
queue-youarenext	Filename of prompt to play	If not defined, plays the default value ("You are now first in line"). If set to an empty value, prompt will not be played at all.
queue-thereare	Filename of prompt to play	If not defined, plays the default value ("There are"). If set to an empty value, prompt will not be played at all.
queue-callswaiting	Filename of prompt to play	If not defined, plays the default value ("calls waiting"). If set to an empty value, prompt will not be played at all.
queue-holdtime	Filename of prompt to play	If not defined, plays the default value ("The current estimated hold time is"). If set to an empty value, prompt will not be played at all.
queue-minutes	Filename of prompt to play	If not defined, plays the default value ("minutes"). If set to an empty value, prompt will not be played at all.
queue-seconds	Filename of prompt to play	If not defined, plays the default value ("seconds"). If set to an empty value, prompt will not be played at all.
queue-reporthold	Filename of prompt to play	If not defined, plays the default value ("hold time"). If set to an empty value, prompt will not be played at all.
periodic-announce	A set of periodic announcements to be played, separated by commas	Prompts are played in the order they are defined. Defaults to queue-periodic-announce ("All representatives are currently busy assisting other callers. Please wait for the next available representative").

There's a ton of flexibility possible when designing a caller's experience while they're waiting, but please don't forget that your callers will never be happy to be waiting in the queue. Also, if you've found some half-decent hold music, and your callers are enjoying it, an interruption to play yet another message runs the risk of really setting their blood boiling. When they are finally answered, your poor agents will get the brunt of their anger, even though it is actually your fault.[5]

So keep your on-hold tweaking simple. Callers know they're waiting, and they aren't going to be happy about it. Get them to an agent as quickly as possible, with the bare minimum amount of silliness while they're holding, and don't succumb to the temptation of making the queue more important to your callers than it actually is.

Overflow

Unfortunately, your queue will not always get your callers to an agent in a timely manner. When various conditions cause the queue to reject incoming callers, we have an overflow situation. Overflowing out of the queue is done either with a timeout value or when no queue members are available (as defined by `joinempty` or `leavewhenempty`). In this section we'll discuss how to control when overflow happens.

Controlling timeouts

The `Queue()` application supports two kinds of timeout: one defines the maximum period of time a caller stays in the queue, and the other specifies how long to ring a device when attempting to connect a caller to a queue member. The two are unrelated but can affect each other. In this section we'll be talking about the maximum period of time a caller stays in the `Queue()` application before the call overflows to the next step in the dialplan, which could be something like `VoiceMail()`, or even another queue. Once the call has fallen out of the queue, it can go anywhere that a call could normally go when controlled by the dialplan.

The timeouts are specified in two locations. The timeout that indicates how long to ring queue members for is specified in the `queues` table. The absolute timeout (how long the caller stays in the queue) is controlled via the `Queue()` application. To set a maximum amount of time for callers to stay in a queue, simply specify it after the queue name in the `Queue()` application:

```
; Queue
exten => 610,1,Noop()
  same => n,Progress()
  same => n,Queue(sales,120)
  same => n,Voicemail(${EXTEN}@queues,u)
  same => n,Hangup()
```

5 Just sayin'.

```
exten => 611,1,Noop()
  same => n,Progress()
  same => n,Queue(support,120)
  same => n,Voicemail(${EXTEN}@queues,u)
  same => n,Hangup()

exten => 612,1,Noop()
  same => n,Progress()
  same => n,Queue(support-priority,120)
  same => n,Voicemail(${EXTEN}@queues,u)
  same => n,Hangup()
```

Since we're sending the calls to voicemail, we'll need some mailboxes:

```
MySQL> INSERT INTO `asterisk`.`voicemail`
(context,mailbox,password,fullname,email)

VALUES
('queues','610','192837','Queue sales','name@shifteight.org'),
('queues','611','192837','Queue support','name@shifteight.org'),
('queues','612','192837','Queue support-priority','name@shifteight.org');
```

Of course, we could define a different destination, but the `VoiceMail()` application is a common overflow destination for a queue. Obviously, sending callers to voicemail is not ideal (they were hoping to speak to someone live), so make sure someone checks it regularly and calls your customers back.

Now, let's say we have set our absolute timeout to 10 seconds, our timeout value for ringing queue members to 5 seconds, and our retry timeout value to 4 seconds. In this scenario, we would ring the queue member for 5 seconds, then wait 4 seconds before attempting another queue member. That brings us up to 9 seconds of our absolute timeout of 10 seconds. At this point, should we ring the second queue member for 1 second and then exit the queue, or should we ring this member for the full 5 seconds before exiting?

We control which timeout value has priority with the `timeoutpriority` option in the `queues` table. The available values are `app` (the default) and `conf`. If we want the application timeout (the absolute timeout) to take priority, which would cause our caller to be kicked out after exactly 10 seconds (even though it was just starting to ring an agent), we should set the `timeoutpriority` value to `app`. If we want the configuration file timeout to take priority and finish ringing the queue member, which will cause the caller to stay in the queue a little longer, we should set `timeoutpriority` to `conf`. The default value is `app` (which is the default behavior in previous versions of Asterisk). Probably in most cases you'll want to use `conf` (especially if you want your caller experience to be as non-weird as possible).

```
MySQL> update `asterisk`.`queues` set timeoutpriority='conf'
  where name in ('sales','support','support-priority');
```

The goal is to get callers to agents, yes?

Controlling when to join and leave a queue

Asterisk provides two options that control when callers can join and are forced to leave queues, both based on the statuses of the queue members. The first option, `joinempty`, is used to control whether callers can enter a queue in the first place. The second option, `leavewhenempty`, is used to control events that will cause callers already in a queue to be removed from that queue (i.e., if all of the queue members become unavailable). Both options allow for a comma-separated list of values to control this behavior, as listed in Table 12-5.

Table 12-5. Options that can be set for joinempty or leavewhenempty

Value	Description
`paused`	Members are considered unavailable if they are paused.
`penalty`	Members are considered unavailable if their penalties are less than `QUEUE_MAX_PENALTY`.
`inuse`	Members are considered unavailable if their device status is `InUse`.
`ringing`	Members are considered unavailable if their device status is `Ringing`.
`unavailable`	Applies primarily to agent channels; if the agent is not logged in but is a member of the queue, the channel is considered unavailable.
`invalid`	Members are considered unavailable if their device status is `Invalid`. This is typically an error condition.
`unknown`	Members are considered unavailable if device status is unknown.
`wrapup`	Members are considered unavailable if they are currently in the wrapup time after the completion of a call.

For `joinempty`, prior to placing a caller into the queue, all the members are checked for availability using the factors you list as criteria. If all members are deemed to be unavailable, the caller will not be permitted to enter the queue, and dialplan execution will continue at the next priority.[6] For the `leavewhenempty` option, the members' statuses are checked periodically against the listed conditions; if it is determined that no members are available to take calls, the caller is removed from the queue, with dialplan execution continuing at the next priority.

An example use of `joinempty` could be:

```
joinempty=unavailable,invalid,unknown
```

With this configuration, prior to a caller entering the queue the statuses of all queue members will be checked, and the caller will not be permitted to enter the queue unless at least one queue member is found to have a status that is not `unavailable`, `invalid`, or `unknown`.

The `leavewhenempty` example could be something like:

6 If the priority *n*+1 (from where the `Queue()` application was called) is not defined, the call will be hung up. In other words, don't use this functionality unless your dialplan does something useful at the step immediately following `Queue()`.

```
leavewhenempty=unavailable,invalid,unknown
```

In this case, the queue members' statuses will be checked periodically, and callers will be removed from the queue if no queue members can be found who do not have a status of `unavailable`, `invalid`, or `unknown`.

Previous versions of Asterisk used the values `yes`, `no`, `strict`, and `loose` as the available values to be assigned. The mapping of those values is shown in Table 12-6.

Table 12-6. Mapping between old and new values for controlling when callers join and leave queues

Value	Mapping (joinempty)	Mapping (leavewhenempty)
`yes`	(empty)	`penalty,paused,invalid`
`no`	`penalty,paused,invalid`	(empty)
`strict`	`penalty,paused,invalid,unavailable`	`penalty,paused,invalid,unavailable`
`loose`	`penalty,invalid`	`penalty,invalid`

Using Local Channels

The use of local channels as queue members is a powerful way of executing dialplan code prior to dialing the actual agent's device. When `Queue()` decides to present a call to an agent, using local channels allows us to define custom channel variables, write to a logfile, set some limit on call length (e.g., if it is a paid service), send messages of all sorts all over the place, perform database transactions, and perform many of the other actions we might wish to do at that exact moment. Normally, we have no control over when the `Queue()` application has decided to present a caller to a specific member, but with local channels, we get one final kick at the can, and can even return `Congestion()`, which will have the effect of returning the caller to the queue, since the queue will not consider this call to have been successfully delivered to an agent (this can be very handy, since some external condition can be evaluated before the call is just fired off to an endpoint).

When using local channels for queues, they are added just like any other channels, typically dynamically through the `AddQueueMember()` dialplan application.

We'll need to define the local channel where all the magic happens, and since local channels are typically used in a manner similar to subroutines, we like to name and locate them in the dialplan with the subroutines, with a context name starting with `local` (akin to how subroutines start with `sub`). If you've been building out your dialplan along with the book, you'll notice you already have a local channel `[local DialDelay]`. Add this code somewhere in that part of the dialplan.

```
[localMemberConnector]
exten => _[A-Za-z0-9].,1,Verbose(2,Connect ${CALLERID(all)} to Agent at ${EXTEN})
   ; filter out any bad characters, allow alphanumeric chars and hyphen
   same => n,Set(QueueMember=${FILTER(A-Za-z0-9\-,${EXTEN})})
```

```
    ; assign the first field of QueueMember to Technology; hyphen as separator
    same => n,Set(Technology=${CUT(QueueMember,-,1)})
    ; assign the second field of QueueMember to Device using the hyphen separator
    same => n,Set(Device=${CUT(QueueMember,-,2)})
    ; dial the agent
    same => n,Dial(${Technology}/${Device})
    same => n,Hangup()
```

This code might not make total sense just yet, but what it's doing is taking the `${EXTEN}` (which is a complex alphanumeric string at this point), and slicing and dicing it to extract the actual channel to be called (i.e., we pass as part of the local channel all the information needed to dial the actual channel).

Let's look at the `AddQueueMember` code and see if we can make more sense of this:

```
exten => *740,1,Noop(Logging in device ${CHANNEL(endpoint)} into the support queue)
   same => n,Set(MemberTech=${CHANNEL(channeltype)})
   same => n,Set(MemberIdent=${CHANNEL(endpoint)})
   same => n,Set(Interface=${MemberTech}/${MemberIdent})
   ;;; THE FOLLOWING SHOULD ALL BE ON ONE LINE
same => n,AddQueueMember(support,Local/${MemberTech}-${MemberIdent}@localMemberConnector
,,,${IF($[${MemberTech} = PJSIP]?${Interface})})
   same => n,Playback(silence/1)
   same => n,Playback(${IF($[${AQMSTATUS} = ADDED]?agent-loginok:agent-incorrect)})
   same => n,Hangup()
```

Once you've input all this and reloaded your dialplan, log into the queue by dialing *740, and let's see what we've got.

```
*CLI> queue show support

support has 0 calls (max unlimited) in 'rrmemory' strategy (1s holdtime, 0s talktime),
W:0, C:1, A:1, SL:0.0%, SL2:0.0% within 0s
   Members:
      PJSIP/SOFTPHONE_A (Local/PJSIP-SOFTPHONE_A@localMemberConnector)
(ringinuse disabled) (dynamic) (Not in use)
   No Callers
```

The member is now identified to the queue as a local channel named `PJSIP-SOFTPHONE_A` in the `[localMemberConnector]` context. (The `PJSIP/SOFTPHONE_A` channel will be monitored for actual status of the endpoint.) When `Queue()` decides to send a call to the member, the call will end up in the `[localMemberConnector]` context, where the `EXTEN` (`PJSIP-SOFTPHONE_A`) will be sliced and diced in order to yield our channel type and endpoint,[7] which is what will actually be called.

At this point, the purpose of all this extra complexity is not immediately clear. So far we don't get anything useful out of all this extra code.

7 Perhaps we could have used `/` instead of `-` as a delimiter, giving us `Local/PJSIP/SOFTPHONE_A@localMemberConnector`, but we felt that would be more prone to strange syntax errors, and awkward to filter and parse, so we went with `-`.

So now that we can add devices to the queue using local channels, let's look at how this might be useful.

Let's say we have a customer who just can't stand our best agent. They're a good customer, so we don't want to lose them, but it's our best agent, so we're not going to fire them.

To set this up, we're going to assign a caller ID to `SOFTPHONE_B`, so we have something to match against.

```
MySQL> UPDATE `asterisk`.`ps_endpoints` SET callerid='SOFTPHONE_B <103>' \
WHERE id='SOFTPHONE_B';
```

We're going to build a little trick into our dialplan that will reject the call to the agent if the caller ID matches our sensitive customer.

```
[localMemberConnector]
exten => _[A-Za-z0-9].,1,Verbose(2,Connect ${CALLERID(all)} to Agent at ${EXTEN})
   same => n,Wait(0.1) ; Prevent loop from completely hogging CPU
   same => n,Set(QueueMember=${FILTER(A-Za-z0-9\-_,${EXTEN})}) ; allow alphanum, - , _
   same => n,Set(Technology=${CUT(QueueMember,-,1)}) ; first field, hyphen is separator
   same => n,Set(Device=${CUT(QueueMember,-,2)})     ; second field, hypen separator
   ; is this our mismatched pair?
   same => n,DumpChan()
   same => n,Noop(${CALLERID(all)} : ${Device})
same=>n,GotoIf($["${CALLERID(num)}"="103"&"${Device}"="SOFTPHONE_A"]?rejectcall:ringagent)
   ; dial the agent
   same => n(ringagent),Dial(${Technology}/${Device})
   same => n,Hangup()
   ; send it back!
   same => n(rejectcall),Congestion()
   same => n,Hangup()
```

The passing back of `Congestion()` will cause the caller to be returned to the queue (while this is happening, the caller gets no indication that anything is amiss and keeps hearing music until their call is answered by a channel of some sort).[8] Ideally, your queue is programmed to try another agent; however, you need to keep in mind that if `app_queue` determines that this member is still its first choice to present the call to, the call will simply be reconnected to the same agent (and get congestion again, and thus potentially create a CPU-hogging logic loop). To avoid this, you will need to ensure your queue is using a distribution strategy such as `round_robin`, `random`, or any strategy that ensures the same member is not tried over and over. This is also why we toss a tiny little delay into our `[localMemberConnector]` context, so if a loop like this does happen, there's at least a small throttle on it.

Let's just sanity check our code. Set the caller ID number to something other than `103`, and the call should go through.

8 Obviously, don't use any dialplan code in your local channel that will answer, such as `Answer()`, `Playback()`, and so forth.

```
MySQL> UPDATE `asterisk`.`ps_endpoints` SET callerid='SOFTPHONE_B <123>' \
WHERE id='SOFTPHONE_B';
```

The use of local channels for your member channels will not make queue design and debugging easier, but it does give you far more power over your queues than just using `app_queue` on its own, so if you have a complex queue requirement, the use of local channels will give you a level of control you would not have otherwise.

Queue Statistics: The queue_log File

The *queue_log* file (commonly located in */var/log/asterisk*) contains cumulative event information for the queues defined in your system (such as when a queue is reloaded, when queue members are added or removed, pause/unpause events, and so forth) as well as some call details (e.g., their status and which channels the callers were connected to). The queue log is enabled by default, but it can be controlled via the */etc/asterisk/logger.conf* file. There are three options related to the *queue_log* file specifically:

`queue_log`
: Controls whether the queue log is enabled or not. Valid values are `yes` or `no` (defaults to `yes`).

`queue_log_to_file`
: Controls whether the queue log should be written to a file even when a real-time backend is present. Valid values are `yes` or `no` (defaults to `no`).

`queue_log_name`
: Controls the name of the queue log. The default is `queue_log`.

The queue log is a pipe-separated list of events. The fields in the *queue_log* file are as follows:

- UNIX Epoch timestamp of the event
- Unique ID of the call
- Name of the queue
- Name of bridged channel
- Type of event
- Zero or more event parameters

The information contained in the event parameters depends on the type of event. A typical *queue_log* file will look something like the following:

```
1530389309|NONE|NONE|NONE|QUEUESTART|
1530409313|CLI|support|PJSIP/SOFTPHONE_B|ADDMEMBER|
1530409467|CLI|support|PJSIP/SOFTPHONE_B|REMOVEMEMBER|
1530409666|NONE|support|PJSIP/SOFTPHONE_B|PAUSE|Callbacks
1530411108|NONE|support|PJSIP/SOFTPHONE_B|UNPAUSE|FinishedCallbacks
```

```
1530440239|1530440239.10|support|PJSIP/SOFTPHONE_A|ADDMEMBER|
1530440303|1530440303.16|support|PJSIP/SOFTPHONE_A|REMOVEMEMBER|
1530497165|1530497165.54|support|Local/PJSIP-SOFTPHONE_A@MemberConnector|ADDMEMBER|
1530497388|CLI|support|Local/PJSIP-SOFTPHONE_A@MemberConnector|REMOVEMEMBER|
1530497408|1530497408.60|support|Local/PJSIP-SOFTPHONE_A@localMemberConnector|ADDMEMBER|
1530497506|1530497506.71|support|NONE|ENTERQUEUE||SOFTPHONE_B|1
1530497511|1530497506.71|support|PJSIP/SOFTPHONE_A|CONNECT|5|1530497506.72|4
1530497517|1530497506.71|support|PJSIP/SOFTPHONE_A|COMPLETEAGENT|5|6|1
1530509861|1530509861.134|support|NONE|ENTERQUEUE||SOFTPHONE_B|1
1530509864|1530509861.134|support|PJSIP/SOFTPHONE_A|RINGCANCELED|2224
1530509864|1530509861.134|support|NONE|ABANDON|1|1|3
1530510503|1530510503.156|support|NONE|ENTERQUEUE||103|1
1530510503|1530510503.156|support|PJSIP/SOFTPHONE_A|RINGNOANSWER|0
1530510511|1530510503.156|support|NONE|ABANDON|1|1|8
1530510738|1530510738.163|support|NONE|ENTERQUEUE||123|1
1530510742|1530510738.163|support|PJSIP/SOFTPHONE_A|CONNECT|4|1530510738.164|4
1530510752|1530510738.163|support|PJSIP/SOFTPHONE_A|COMPLETECALLER|4|10|1
```

As you can see from this example, there may not always be a unique ID for the event. External services, such as the Asterisk CLI, can perform actions on the queue, and in these cases you'll see something like `CLI` in the Unique ID field.

The available events and the information they provide are described in Table 12-7.

Table 12-7. Events in the Asterisk queue log

Event	Information provided
ABANDON	Written when a caller in a queue hangs up before his call is answered by an agent. Three parameters are provided for ABANDON: the position of the caller at hangup, the original position of the caller when entering the queue, and the amount of time the caller waited prior to hanging up.
ADDMEMBER	Written when a member is added to the queue. The bridged channel name will be populated with the name of the channel added to the queue.
AGENTDUMP	Indicates that the agent hung up on the caller while the queue announcement was being played, prior to them being bridged together.
AGENTLOGIN	Recorded when an agent logs in. The bridged channel field will contain something like `Agent/9994` if logging in with `chan_agent`, and the first parameter field will contain the channel logging in (e.g., `SIP/0000FFFF0001`).
AGENTLOGOFF	Logged when an agent logs off, along with a parameter indicating how long the agent was logged in for. Note that since you will often use `RemoveQueueMember()` for agent log off, this parameter may not be written. See the REMOVEMEMBER event instead.
COMPLETE AGENT	Recorded when a call is bridged to an agent and the agent hangs up, along with parameters indicating the amount of time the caller was held in the queue, the length of the call with the agent, and the original position at which the caller entered the queue.
COMPLETECALLER	Same as COMPLETEAGENT, except the caller hung up and not the agent.
CONFIGURE LOAD	Indicates that the queue configuration was reloaded (e.g., via *module reload app_queue.so*).
CONNECT	Written when the caller and the agent are bridged together. Three parameters are also written: the amount of time the caller waited in the queue, the unique ID of the queue member's channel to which the caller was bridged, and the amount of time the queue member's phone rang prior to being answered.

Event	Information provided
ENTERQUEUE	Written when a caller enters the queue. Two parameters are also written: the URL (if specified) and the caller ID of the caller.
EXITEMPTY	Written when the caller is removed from the queue due to a lack of agents available to answer the call (as specified by the `leavewhenempty` parameter). Three parameters are also written: the position of the caller in the queue, the original position at which the caller entered the queue, and the amount of time the caller was held in the queue.
EXITWITHKEY	Written when the caller exits the queue by pressing a single DTMF key on his phone to exit the queue and continue in the dialplan (as enabled by the `context` parameter in *queues.conf*). Four parameters are recorded: the key used to exit the queue, the position of the caller in the queue upon exit, the original position the caller entered the queue at, and the amount of time the caller was waiting in the queue.
EXITWITHTIMEOUT	Written when the caller is removed from the queue due to timeout, as specified by the `timeout` parameter to `Queue()`. Three parameters are also recorded: the position the caller was in when exiting the queue, the original position of the caller when entering the queue, and the amount of time the caller waited in the queue.
PAUSE	Written when a queue member is paused.
PAUSEALL	Written when all members of a queue are paused.
UNPAUSE	Written when a queue member is unpaused.
UNPAUSEALL	Written when all members of a queue are unpaused.
PENALTY	Written when a member's penalty is modified. The penalty can be changed through several means, such as the `QUEUE_MEMBER_PENALTY()` function, the Asterisk Manager Interface, or the Asterisk CLI commands.
REMOVEMEMBER	Written when a queue member is removed from the queue. The bridge channel field will contain the name of the member removed from the queue.
RINGNOANSWER	Logged when a queue member is rung for a period of time, and the timeout value for ringing the queue member is exceeded. A single parameter will also be written indicating the amount of time the member's extension rang.
TRANSFER	Written when a caller is transferred to another extension. Additional parameters are also written, which include the extension and context the caller was transferred to, the hold time of the caller in the queue, the amount of time the caller was speaking to a member of the queue, and the original position of the caller when he entered the queue.[a]
SYSCOMPAT	Recorded if an agent attempts to answer a call, but the call cannot be set up due to incompatibilities in the media setup.

[a] Please note that when the caller is transferred using SIP transfers (rather than the built-in transfers triggered by DTMF and configured in *features.conf*), the TRANSFER event may not be written.

Conclusion

We started this chapter with a look at basic call queues, discussing what they are, how they work, and when you might want to use one. After building a simple queue, we explored how to control queue members through various means (including the use of local channels, which provide the ability to perform some dialplan logic just prior to connecting to a queue member). Of course, we need the ability to monitor what our

queues are doing, so we had a quick look at the *queue_log* file, and the various fields written as a result of events happening in our queues.

With the information provided in this chapter, you have most of the foundational knowledge required to implement queues in Asterisk.

CHAPTER 13
Device States

Out of clutter, find simplicity.
—Albert Einstein

It is often useful to be able to determine the state of the devices that are attached to a telephone system. For example, a receptionist might require the ability to see the statuses of everyone in the office in order to determine whether somebody can take a phone call. Asterisk itself needs this same information. As another example, if you were building a call queue, as discussed in Chapter 12, Asterisk needs to know when an agent is available so that another call can be delivered. This chapter discusses device state concepts in Asterisk, as well as how devices and applications use and access this information.

Device States

There are two categories of devices that Asterisk provides state information for: channel devices (such as PJSIP endpoints) and virtual devices (which are built-in services that one might wish to monitor, such as conference rooms).

To reference the state of a *channel*, you do so in exactly the same way you would with `Dial()`, for example `DEVICE_STATE(PJSIP/000f300B0B02)`, whereas to reference the state of a *virtual device*, the format is *`virtual device type`*`:`*`identifier`*, for example `DEVICE_STATE(ConfBridge:1234)`.

Virtual devices include things that are inside Asterisk but provide useful state information (see Table 13-1).

Table 13-1. Devices for which Asterisk can provide state information

Device	Description
`PJSIP/channel name`	Many channels can have their state monitored, but the PJSIP channel offers by far the most amount of useful data; thus, monitoring SIP devices is the most common use of `DEVICE_STATE`.
`ConfBridge:conference bridge`	The state of a MeetMe conference bridge. The state will reflect whether or not the conference bridge currently has participants called in. More information on using `MeetMe()` for call conferencing can be found in Chapter 11.
`Custom:custom name`	Custom device states. These states have custom names and are modified using the `DEVICE_STATE()` function. Example usage can be found in "Using Custom Device States" on page 244.
`Park:exten@context`	The state of a spot in a call parking lot. The state information will reflect whether or not a caller is currently parked at that extension. More information about call parking in Asterisk can be found in "Call Parking" on page 195.
`Calendar:calendar name`	Calendar state. Asterisk will use the contents of the named calendar to set the state to `available` or `busy`.

Checking Device States

The `DEVICE_STATE()` dialplan function reads the current state of a device.

```
exten => 7012,1,Answer()
 same => n,Set(DeviceIdent=PJSIP/000f300B0B02)
 same => n,Verbose(3,${DeviceIdent} is ${DEVICE_STATE($DeviceIdent)})
 same => n,Hangup()
```

If we call extension 7012 from the same device that we are checking the state of, the following verbose message comes up on the Asterisk console:

```
-- PJSIP/000f300B0B02 is INUSE
```

Chapter 17 discusses the Asterisk Manager Interface (AMI). The `GetVar` manager action can be used to retrieve device state values in an external program. You can use it to get the value of a normal variable, or to return a value from a dialplan function such as `DEVICE_STATE()`.

Here is a list of the values that the `DEVICE_STATE()` function will return (depending, of course, on what was found):

- `UNKNOWN`
- `NOT_INUSE`
- `INUSE`
- `BUSY`
- `INVALID`

- UNAVAILABLE
- RINGING
- RINGINUSE
- ONHOLD

This information can then be used in the dialplan for call-flow decisions (for example, a local channel ringing an agent might use this information to determine that an agent phone is on a call on another line, and thus reject the call so it passes back into the queue).

Extension States Using the hint Directive

Extension state is a dialplan mechanism Asterisk uses to allow SIP devices to subscribe to presence information. As an example, a reception phone might have a Busy Lamp Field (BLF) module, containing buttons to be used to show the state of various phones in the office. The phone with the BLF will send subscription requests in order to tell Asterisk which devices it wants to receive presence information from. In the dialplan, we use the `hint` directive to define the mapping between an extension and one or more devices.

Hints

To define a hint in the dialplan, the keyword `hint` is used in place of a priority.

```
[hints]
;exten = <extension>,hint,<device state id>[& <more dev state id],<presence state id>

exten => 100,hint,${UserA_DeskPhone}

exten => 221,hint,ConfBridge:221
```

Often, you might see hints defined in the same section of the dialplan as the normal extension. This can get a bit visually cluttered, and it also suggests that the hint is somehow associated with the dialable extension, which is not the case.

```
[sets]

exten => 100,hint,${UserA_DeskPhone}
exten => 100,1,Gosub(subDialUser,${EXTEN},1(${UserA_DeskPhone},${EXTEN},default,22))
exten => 101,hint,${UserA_SoftPhone}
exten => 101,1,Gosub(subDialUser,${EXTEN},1(${UserA_SoftPhone},${EXTEN},default,23))
exten => 102,hint,${UserB_DeskPhone}
exten => 102,1,Gosub(subDialUser,${EXTEN},1(${UserB_DeskPhone},${EXTEN},default,26))
exten => 103,hint,${UserB_SoftPhone}
exten => 103,1,Gosub(subDialUser,${EXTEN},1(${UserB_SoftPhone},${EXTEN},default,24))

exten => 110,1,Dial(${UserA_DeskPhone}&${UserA_SoftPhone}&${UserB_SoftPhone})
```

In our example we've made a direct correlation between the hint's extension number and the extension number being dialed, although there is no requirement that this be the case.

Checking Extension States

The easiest way to check the current state of the hint extensions is through the Asterisk CLI. The `core show hints` command will display all currently configured hints:

```
*CLI> core show hints
    -= Registered Asterisk Dial Plan Hints =-
100@hints : PJSIP/0000f30A0A01   State:Unavailable Presence:not_set Watchers 0
101@hints : PJSIP/SOFTPHONE_A    State:Unavailable Presence:not_set Watchers 0
102@hints : PJSIP/0000f30B0B02   State:Unavailable Presence:not_set Watchers 0
103@hints : PJSIP/SOFTPHONE_B    State:Unavailable Presence:not_set Watchers 0
110@hints : PJSIP/0000f30A0A01&P State:Unavailable Presence:not_set Watchers 0
221@hints : ConfBridge:221       State:Unavailable Presence:not_set Watchers 0
----------------
- 6 hints registered
```

In addition to showing you the state of each hint, the output of `core show hints` also provides a count of watchers. A *watcher* is an entity that has subscribed to receive updates on the state of this extension. If a SIP endpoint subscribes to the state of an extension, the watcher count will be increased.

Extension state can also be retrieved with a dialplan function, `EXTENSION_STATE()`. This function operates much like the `DEVICE_STATE()` function described in the preceding section. Add the following example to your */etc/asterisk/extensions.conf* file, as a new extension right after 235:

```
exten => 234,1,NoOp()
  same => n,Set(FEATURE(parkingtime)=60)

exten => 235,1,Noop(The state of 100@hints is ${EXTENSION_STATE(100@hints)} )
  same => n,Hangup()

exten => 321,1,NoOp()
```

When this extension is called from the endpoint assigned to 100, this is the message that shows up on the Asterisk console:

```
    -- The state of 100@hints is INUSE
```

The following list includes the possible values that may be returned from the `EXTENSION_STATE()` function:

- `UNKNOWN`
- `NOT_INUSE`
- `INUSE`
- `BUSY`

- `UNAVAILABLE`
- `RINGING`
- `RINGINUSE`
- `HOLDINUSE`
- `ONHOLD`

SIP Presence

Asterisk gives devices the capability to subscribe to extension state using the SIP protocol. This functionality is often referred to as BLF (Busy Lamp Field); see Figure 13-1.

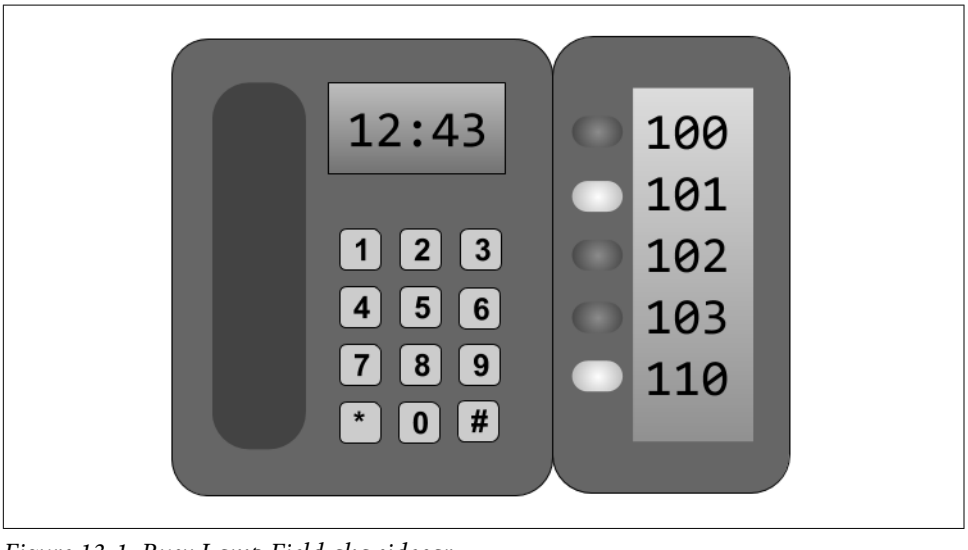

Figure 13-1. Busy Lamp Field aka sidecar

The configuration of the module will be slightly (or very) different for each manufacturer; however, the subscription information will—one way or another—need to include the following:

- The address of the Asterisk server (this might be defined on a per-button basis, or it might apply to the whole phone).
- The context to subscribe to (in our sample dialplan, it's named `[hints]`). This setting is defined in the `subscribe_context` field of the `asterisk.ps_endpoints` table.

- The relevant extension (100, 101, 102, etc.)[1]

One of the more simple and inexpensive ways we've found for testing presence is using the open source Windows softphone, MicroSIP.[2] You'll first need to download MicroSIP and get it registered to your Asterisk system. Then, under the contacts tab of the softphone, you can right-click in the open area to Add a contact. In the Name section you can put whatever you wish, but under the Number section, you will input *`extension@hints context`*, which in our case would be one of `100@hints`, `101@hints`, `102@hints`, or `103@hints`. If you've set everything up in Asterisk per the previous examples, you should see the state of your subscriptions change in response to whatever the far end set is doing. You can also monitor this from Asterisk's perspective using a command such as:

```
$ watch -n 0.5 "sudo asterisk -rx 'core show hints'"
```

The configuration of presence on physical desk telephones is essentially the same, but it can be more difficult to make sense of the specific syntax each manufacturer requires. Our advice is to get it working with MicroSIP (which you should be able to run on WINE under Linux or macOS). It's an easy setup, and from there you'll have a known-good configuration you can trust when you're sorting out a similar config for one of your desk phones.

Using Custom Device States

In addition to the devices Asterisk knows internally how to monitor (`PJSIP`, `Conf Bridge`, `Park`, `Calendar`), Asterisk also provides the ability to create custom device states, which can be very useful in the development of some interesting applications.

Custom device states are defined using a prefix of `Custom:`. The text that comes after the prefix can be anything you want. To set or read the value of a custom device state, use the `DEVICE_STATE()` dialplan function. Put this into your *extensions.conf* right after extension 235:

```
exten => 235,1,Noop(The state of 100@hints is ${EXTENSION_STATE(100@hints)} )
  same => n,Hangup()

exten => 236,1,Noop(Set a custom status)
  same => n(blink),Set(DEVICE_STATE(Custom:rudolph)=UNAVAILABLE)
  same => n,Set(DEVICE_STATE(Custom:santa)=NOT_INUSE)
  same => n,Wait(0.75)
  same => n,Set(DEVICE_STATE(Custom:rudolph)=NOT_INUSE)
  same => n,Set(DEVICE_STATE(Custom:santa)=UNAVAILABLE)
  same => n,Wait(0.75)
  same => n,Goto(blink)
```

1 Items 2 and 3 may be formed as a single string, looking like `100@hints`, or something similar.

2 Which is written using the same PJSIP library that Asterisk uses.

Then add this to your [hints] context:

```
exten => 221,hint,ConfBridge:221
exten => santa,hint,Custom:santa
exten => rudolph,hint,Custom:rudolph
```

Festive, yeah?

You will notice that when you hang up, one of the custom device states will remain "Unavailable." This is an important point: there is nothing in the system that will update your custom device states, unless you yourself have implemented something to do that.

Conclusion

The device states functionality in Asterisk can be used to track the state of various resources and deliver information about those states to various subscribers. Commonly (and traditionally) used for Busy Lamp Fields, the Custom device state allows this resource to be far more flexible than it would be in a traditional PBX.

CHAPTER 14
The Automated Attendant

> *I don't answer the phone. I get the feeling whenever I do that there will be someone on the other end.*
>
> —Fred Couples

In many PBXs, it is common to have a menu system in place to answer incoming calls automatically and allow callers to direct themselves to various extensions and resources in the system through menu choices. This is known in the telecom industry as an *automated attendant* (AA). An AA normally provides the following features:

- Transfer to extension
- Transfer to voicemail
- Transfer to a queue
- Play message (e.g., "our address is…")
- Connect to a submenu (e.g., "for a listing of our departments...")
- Connect to reception
- Repeat choices

For anything else—especially if there is external integration required, such as a database lookup—an Interactive Voice Response (IVR) would normally be needed.

An AA Is Not an IVR

In the open source telecom community, you will often hear the term *IVR* used to describe an automated attendant.[1] However, in the telecom industry, for many decades before there was VoIP or open source PBXs, an IVR was distinct from an AA. For this reason, when you are talking to somebody with many years of telecom experience about any sort of telecom menu, you should ensure that you are talking about the same thing. To a telecom professional, the term *IVR* implies a relatively complex and involved development effort (and subsequent costs), whereas an AA is a simple and inexpensive thing that is common to most PBXs.

In this chapter, we talk about building an automated attendant. In Chapter 16 we will discuss IVR.[2]

Designing Your AA

The most common mistake beginners make when designing an AA is needless complexity. While there can be much joy and sense of accomplishment in the creation of a multilevel AA with dozens of nifty options and oodles of really cool prompts, your callers have a different agenda. The reason people make phone calls is primarily because they want to talk to someone. While people have become used to the reality of automated attendants (and in some cases they can speed things up), for the most part people would prefer to speak to somebody live. This means that there are two fundamental rules that every AA should adhere to:

- Keep it simple.
- Make sure you always include a handler for the folks who are going to press 0 whenever they hear a menu. If you do not want to have a 0 option, be aware that many people will be insulted by this, and they will hang up and not call back. In business, this is generally a bad thing.

Before you start to code your AA, it is wise to design it. You will need to define a call flow, and you will need to specify the prompts that will play at each step. Software diagramming tools can be useful for this, but there's no need to get fancy. Table 14-1 provides a good template for a basic AA that will do what you need.

1 This is most likely because "IVR" is much easier to say than "automated attendant."

2 It should be noted that Asterisk is an excellent IVR-creation tool. It's not bad for building automated attendants, either.

Table 14-1. A basic automated attendant

Step or choice	Sample prompt	Notes	Filename[a]
Greeting—business hours	Thank you for calling ABC company.	Day greeting. Played immediately after the system answers the call.	*daygreeting.wav*
Greeting—nonbusiness hours	Thank you for calling ABC company. Our office is now closed.	Night greeting. As above, but plays outside of business hours.	*nightgreeting.wav*
Main menu	If you know the extension of the person you wish to reach, please enter it now. For sales, please press 1; for service, press 2; for our company directory, press #. For our address and fax information, please press 3. To repeat these choices press 9, or you can remain on the line or press 0 to be connected to our operator.	Main menu prompt. Plays immediately after the greeting. For the caller, the greeting and the main menu are heard as a single prompt; however, in the system it is helpful to keep these prompts separate.	*mainmenu.wav*
`1`	Please hold while we connect your call.	Transfer to sales queue.	*holdwhileweconnect.wav*
`2`	Please hold while we connect your call.	Transfer to support queue.	*holdwhileweconnect.wav*
`#`	n/a	Run `Directory()` application	n/a
`3`	Our address is [address]. Our fax number is [fax number], etc.	Play a recording containing address and fax information. Return caller to menu prompt when done.	*faxandaddress.wav*
`0`	Transferring to our attendant. Please hold.	Transfer to reception/operator.	*transfertoreception.wav*
`9`	n/a	Repeat. Replay menu prompt (but not greeting).	n/a
`t`	n/a	Timeout. If the caller does not make a choice, treat the call as if caller has dialed 0 (or in some cases, replay the prompt).	
`i`	You have made an invalid selection. Please try again.	Caller pressed an invalid digit: replay menu prompt (but not greeting).	*invalid.wav*
`_XXX` [b]	n/a	Transfer call to dialed extension.	*holdwhileweconnect.wav*

[a] These files don't exist anywhere as of yet. We're using these as examples.

[b] This pattern match must be relevant to your extension range.

Let's go over the various components of this template. Then we'll show you the dialplan code required to implement it, as well as how to create prompts for it.

The Greeting

The first thing the caller hears is actually two prompts.

The first prompt is the greeting. The only thing the greeting should do is greet the caller. Examples of a greeting might be "Thank you for calling Bryant, Van Meggelen, and Associates," "Welcome to Leif's School of Wisdom and T-Shirt Design," or "You have reached the offices of Dewey, Cheetum, and Howe, Attorneys." That's it—the choices for the caller will come later. This allows you to record different greetings without having to record a whole new menu. For example, for a few weeks each year you might want your greeting to say "Season's greetings" or whatever, but your menu will not need to change. Also, if you want to play a different recording after hours ("Thank you for calling. Our office is now closed."), you can use different greetings, but the heart of the menu can stay the same. Finally, if you want to be able to return callers to the menu from a different part of the system, you will normally not want them to hear the greeting again.

The Main Menu

The main menu prompt is where you inform your callers of the choices available to them. You should speak this as quickly as possible (without sounding rushed or silly).[3] When you record a choice, *always tell the users the action that will be taken before giving them the digit option to take that action*. So, don't say "press 1 for sales," but rather say "for sales, press 1." The reason for this is that most people will not pay full attention to the prompt until they hear the choice that is of interest to them. Once they hear their choice, you will have their full attention and can tell them what button to press to get them to where they want to go.

Another point to consider is what order to put the choices in. A typical business, for example, will want sales to be the first menu choice, and most callers will expect this as well. The important thing is to think of your callers. For example, most people will not be interested in address and fax information, so don't make that the first choice.[4] Think about the goal of getting the callers to their intended destinations as quickly as possible when you make your design choices. Ruthlessly cut anything that is not absolutely essential.

3 If necessary, you can use an audio editing program such as Audacity to remove silence, and even to speed up the recording a bit.

4 In fact, we don't normally recommend this in an AA because it adds to what the caller has to listen to, and most people will go to a website for this sort of information.

Selection 1

Option 1 in our example will be a simple transfer. Normally this would be to a resource located in another context, and it would typically have an internal extension number so that internal users could also transfer calls to it. In this example, we are going to use this option to send callers to the queue called `sales` that was created in Chapter 12.

Selection 2

Option 2 will be technically identical to option 1. Only the destination will be different. This selection will transfer callers to the `support` queue.

Selection

It's good to have the option for the directory as close to the beginning of the recording as possible. Many people will use a directory if they know it is there, but can't be bothered to listen to the whole menu prompt to find out about it. Impatient people will press 0, so the sooner you tell them about the directory, the better the odds that they'll use it, and thus reduce the workload on your receptionist.

Selection 3

When you have an option that does nothing but play a recording back to the caller (such as address and fax information), you can leave all the code for that in the same context as the menu, and simply return the caller to the main menu prompt at the end of the recording. In general, these sorts of options are not as useful as we would like to think they are, so in most cases you'll probably want to leave this out.

Selection 9

It is very important to give the caller the option to hear the choices again. Many people will not be paying attention throughout the whole menu, and if you don't give them the option to hear the choices again, they will most likely press 0.

Note that you do not have to play the greeting again, only the main menu prompt.

Selection 0

As stated before, and whether you like it or not, this is the choice that many (possibly the majority) of your callers will select. If you really don't want to have somebody handle these calls, you can send this extension to a mailbox, but we don't recommend it. If you are a business, many of your callers will be your customers. You want to make it easy for them to get in touch with you. Trust us.

Timeout

Many people will call a number and not pay too much attention to what is happening. They know that if they just wait on the line, they will eventually be transferred to the operator. Or perhaps they are in their cars, and really shouldn't be pressing buttons on their phones. Either way, oblige them. If they don't make any selection, don't harass them and force them to do so. Connect them to the operator.

Invalid

People make mistakes. That's OK. The invalid handler will let them know that whatever they have chosen is not a valid option and will return them to the menu prompt so that they can try again. Note that you should not play the greeting again, only the main menu prompt.

Dial by Extension

If somebody calls your system and knows the extension she wants to reach, your automated attendant should have code in place to handle this.

Although Asterisk can handle an overlap between menu choices and extension numbers (e.g., you can have a menu choice 1 and extensions from 100 to 199), it is generally best to avoid this overlap. Otherwise, the dialplan will always have to wait for the interdigit timeout whenever somebody presses 1, because it won't know if they are planning to dial extension 123. The interdigit timeout is the delay the system will allow between digits before it assumes the entire number has been input. This timer ensures callers have enough time to dial a multidigit extension, but it also causes a delay in the processing of single-digit inputs.

Building Your AA

After you have designed your AA, there are three things you need to do to make it work properly:

- Record prompts.
- Build the dialplan for the menu.
- Direct the incoming channels to the AA context.

We will start by talking about recordings.

Recording Prompts

Recording prompts for a telephone system is a critical task. This is what your callers will hear when they interact with your system, and the quality and professionalism of these prompts will reflect on your organization.

Asterisk is very flexible in this regard and can work with many different audio formats. We have found that, in general, the most useful format to use is WAV. Files saved in this format can be of many different kinds, but only one type of WAV file will work with Asterisk: files must be encoded in 16-bit, 8,000 Hz, mono format.

Recommended Prompt File Format

The WAV file format we have recommended is useful for system prompts because it can easily be converted to any other format that your phones might use, without loss or distortion, and almost any computer can play it without any special software. Thus, not only can Asterisk handle the file easily, but it is also easy to work with it on a PC (which can be useful). Asterisk can handle other file formats as well, and in some cases these may be more suitable for your needs, but in general we find 16-bit 8 kHz WAV files to be the easiest to work with and, most of the time, the best possible quality.

There are essentially two ways to get prompts into a system. One is to record sound files in a studio or on a PC, and then move those files into the system. A second way is to record the prompts directly onto the system using a telephone set. We prefer the second method.

Our advice is this: don't get hung up on the complexities of recording audio through a PC or in a studio.[5] It is generally not necessary. A telephone set will produce excellent-quality recordings, and the reasons are simple: the microphone and electronics in a telephone are carefully designed to capture the human voice in a format that is ideal for transmission on telephone networks, and therefore a phone set is also ideal for doing prompts. The set will capture the audio in the correct format, and will filter out background noise and normalize the decibel level.

Yes, a properly produced studio prompt will be superior to a prompt recorded over a telephone, but if you don't have the equipment or experience, take our advice and use a telephone to do your recordings, because a poorly produced studio prompt will be much worse than a prompt recorded through a phone set.

5 Unless you are an expert in these areas, in which case go for it!

Using the dialplan to create recordings

The simplest method of recording prompts is to use the `Record()` application.

Add this new subroutine at the bottom of your *extensions.conf* file:

```
[subRecordPrompt]
exten => 500,1,Playback(vm-intro)
   same => n,Record(daygreeting.wav)
   same => n,Wait(2)
   same => n,Playback(daygreeting)
   same => n,Hangup

exten => 501,1,Playback(vm-intro)
   same => n,Record(mainmenu.wav)
   same => ... etc ... (create dialplan code for each prompt you need to record)
```

In order to use this context, you will need to include it in the context where your sets enter the dialplan. So in your `[LocalSets]` context, you will want to add the line **`include=>UserServices`**. In a production environment, you'll probably want a password on this so that not just anybody can record prompts.

This subroutine plays a prompt, issues a beep, makes a recording, and plays that recording back.[6] It's notable that the `Record()` application takes the entire filename as its argument, while the `Playback()` application excludes the filetype extension (*.wav*, *.gsm*, etc.). This is because the `Record()` application needs to know which format the recording should be made in, while the `Playback()` application does not. Instead, `Playback()` automatically selects the best audio format available, based upon the codec your handset is using and the formats available in the *sounds* folder (for example, if you have a *daygreeting.wav* and a *daygreeting.gsm* file in your *sounds* folder, `Playback(daygreeting)` will select the one that requires the least CPU to play back to the caller).

You'll probably want a separate extension for recording each of the prompts, possibly hidden away from your normal set of extensions, to prevent a mistyped extension from wiping out any of your current menu prompts. If the number of prompts you have is large, repeating this extension with slight modifications for each will get tedious, but there are ways around that. We'll show you how to make your prompt recording more intelligent in Chapter 16, but for now the method just described will serve our immediate needs.

6 The *vm-intro* prompt isn't perfect (it asks you to leave a message), but it's close enough for our purposes. The usage instructions at least are correct: press **#** to end the recording. Once you've gotten the hang of recording prompts, you can go back, record a custom prompt, and change priority 1 to reflect more appropriate instructions for recording your own prompts.

Here's the dialplan (in bold) that'll create all our prompts. Place it wherever you wish in the [sets] context:

```
exten => _4XX,1,Noop(User Dialed ${EXTEN})
  same => n,Answer()
  same => n,SayDigits(${EXTEN})
  same => n,Hangup()

exten => 500,1,GoSub(subRecordPrompt,${EXTEN},1(daygreeting)
exten => 501,1,GoSub(subRecordPrompt,${EXTEN},1(nightgreeting)
exten => 502,1,GoSub(subRecordPrompt,${EXTEN},1(mainmenu)
exten => 503,1,GoSub(subRecordPrompt,${EXTEN},1(holdwhileweconnect)
exten => 504,1,GoSub(subRecordPrompt,${EXTEN},1(faxandaddress)
exten => 505,1,GoSub(subRecordPrompt,${EXTEN},1(transfertoreception)
exten => 506,1,GoSub(subRecordPrompt,${EXTEN},1(invalid)
exten => 507,1,GoSub(subRecordPrompt,${EXTEN},1(holdwhileweconnect)

exten => _555XXXX,1,Answer()
  same => n,SayDigits(${EXTEN})
exten => _55512XX,1,Answer()
  same => n,Playback(tt-monkeys)
```

The recordings (aka prompts) will be placed in the */var/lib/asterisk/sounds* folder. You can put them elsewhere, so long as you specify the full path when recording and playing back (and ensure the directory where you put them is readable by the asterisk user). In a production system, you should put them elsewhere, so as to separate your custom prompts from the generic prompts. For now, we'll keep things simple and put them in the same folder as the system prompts.

The Dialplan

Here is the code required to create the AA that we designed earlier. We will often use blank lines before labels within an extension to make the dialplan easier to read, but note that just because there is a blank line does not mean there is a different extension.

You can place this code at the end of your [TestMenu] context, right before your subroutines:

```
[MainMenu]

exten => s,1,Verbose(1, Caller ${CALLERID(all)} has entered the auto attendant)
    same => n,Answer()

; this sets the inter-digit timer
    same => n,Set(TIMEOUT(digit)=2)

; wait one second to establish audio
    same => n,Wait(1)

; If Mon-Fri 9-5 goto label daygreeting
    same => n,GotoIfTime(9:00-17:00,mon-fri,*,*?daygreeting:afterhoursgreeting)

    same => n(afterhoursgreeting),Background(nightgreeting) ; AFTER HOURS GREETING
```

```
    same => n,Goto(menuprompt)

    same => n(daygreeting),Background(daygreeting)   ; DAY GREETING
    same => n,Goto(menuprompt)

    same => n(menuprompt),Background(mainmenu) ; MAIN MENU PROMPT
    same => n,WaitExten(4)                     ; more than 4 seconds is probably
                                               ; too much
    same => n,Goto(0,1)                        ; Treat as if caller has pressed '0'

exten => 1,1,Verbose(1, Caller ${CALLERID(all)} has entered the sales queue)
    same => n,Goto(sets,610,1)     ; Sales Queue - see Chapter 13 for details

exten => 2,1,Verbose(1, Caller ${CALLERID(all)} has entered the service queue)
    same => n,Goto(sets,611,1)     ; Service Queue - see Chapter 13 for details

exten => 3,1,Verbose(1, Caller ${CALLERID(all)} has requested address and fax info)
    same => n,Background(faxandaddress)            ; Address and fax info
    same => n,Goto(s,menuprompt)      ; Take caller back to main menu prompt

exten => #,1,Verbose(1, Caller ${CALLERID(all)} is entering the directory)
    same => n,Directory(default)   ; Send the caller to the directory.
                                   ; Use InternalSets as the dialing context

exten => 0,1,Verbose(1, Caller ${CALLERID(all)} is calling the operator)
    same => n,Goto(sets,611,1)     ; Service Queue - see Chapter 13 for details

exten => i,1,Verbose(1, Caller ${CALLERID(all)} has entered an invalid selection)
    same => n,Playback(invalid)
    same => n,Goto(s,menuprompt)

exten => t,1,Verbose(1, Caller ${CALLERID(all)} has timed out)
    same => n,Goto(0,1)

; You will want to have a pattern match for the various extensions
; that you'll allow external callers to dial
; BUT DON'T JUST INCLUDE THE LocalSets CONTEXT
; OR EXTERNAL CALLERS WILL BE ABLE TO MAKE CALLS OUT OF YOUR SYSTEM

; WHATEVER YOU DO HERE, TEST IT CAREFULLY TO ENSURE EXTERNAL CALLERS
; WILL NOT BE ABLE TO DO ANYTHING BUT DIAL INTERNAL EXTENSIONS

exten => _1XX,1,Verbose(1,Call to an extension starting with '1')
    same => n,Goto(sets,${EXTEN},1)
```

Delivering Incoming Calls to the AA

Any call coming into the system will enter the dialplan in the context defined for whatever channel the call arrives on. In many cases this will be a context named `[incoming]`, or `[from-pstn]`, or something similar. The calls will arrive either with an extension (as would be the case with a DID) or without one (which would be the case with a traditional analog line).

Whatever the name of the context, and whatever the name of the extension, you will want to send each incoming call to the menu.

```
[incoming] ; a DID coming in on a channel with
           ; context=incoming
exten => 4169671111,1,Goto(MainMenu,s,1)
```

Depending on how you configure your incoming channels, you will generally want to use the `Goto()` application if you want to send the call to an AA. This is far neater than just coding your whole AA in the `incoming` context.

Since we don't have any incoming circuits in our lab,[7] we're going to create a simple extension that'll deliver us to our fancy new AA:

```
exten => 613,1,Noop()
  same => n,Goto(MainMenu,s,1)
  same => n,Hangup()
```

And that's it! A simple automated attendant that is easy to manage, and will handle the expectations of most callers.

IVR

We'll cover Interactive Voice Response (IVR) in more depth in Chapter 16, but before we do that, we're going to talk about something that is essential to any IVR. Database integration is the subject of the next chapter.

Conclusion

An automated attendant can provide a very useful service to callers. However, if it is not designed and implemented well, it can also be a barrier to your callers that may well drive them away. Take the time to carefully plan out your AA, and keep it simple.

7 And if you do in yours, congratulations and please be careful swimming with the sharks.

CHAPTER 15

Relational Database Integration

Few things are harder to put up with than the annoyance of a good example.
—Mark Twain

In this chapter, we are going to explore integrating some Asterisk features and functions into a database. There are several databases available for Linux, and Asterisk supports the most popular of them through its ODBC connector. While this chapter will demonstrate examples using the ODBC connector with a MySQL database, you will find that most of the concepts will apply to any database supported by unixODBC.

Integrating Asterisk with databases is one of the fundamental aspects of building a large clustered or distributed system. The power of the database will enable you to use dynamically changing data in your dialplans, for tasks like sharing information across an array of Asterisk systems or integrating with web-based services. Our favorite dialplan function, which we will cover later in this chapter, is `func_odbc`. We'll also take a look at the Asterisk Realtime Architecture (ARA), call detail records (CDR), and logging details from any ACD queues you might have.

While not all Asterisk deployments will require relational databases, understanding how to harness them opens a treasure chest full of new ways to design your telecom solution.

Your Choice of Database

In Chapter 3, we installed and configured MySQL, plus the ODBC connector to it, and we've been using the tables that Asterisk provides to allow various configuration options to be stored in the database.

We chose MySQL primarily because it is still the most popular open source database engine, and rather than bouncing around, duplicating trivial commands on various different engines, we left implementing other types of databases to the skill set of the reader. If you want to use a different database such as MariaDB, PostGreSQL, Microsoft SQL, or in fact dozens (perhaps hundreds) of other databases supported by unixODBC, it's quite likely that Asterisk will work with it.

Asterisk also offers native connectors to several databases; however, ODBC works so well we've never found any obvious reason to do things any other way. We're going to both recommend ODBC, and also focus exclusively on it. If you have a preference for something else, this chapter should still provide you with the fundamentals, as well as some working examples, and from there you are of course free to branch out into other methodologies.

Note that regardless of the database you choose, this book cannot teach you about databases. We have tried as best as we can to provide examples that do not require too much expertise in database administration (DBA), but the simple fact is that basic DBA skills are a prerequisite for being able to fully harness the power of any database, including any you might wish to integrate with your Asterisk system. Database skills are essential to nearly all system administrative disciplines these days, so we felt it was appropriate to assume at least a basic level of familiarity with database concepts.

Managing Databases

While it isn't within the scope of this book to teach you how to manage databases, it is at least worth noting briefly some of the applications you could use to help with database management. There are many options, some of which are local client applications running from your computer and connecting to the database, and others being web-based applications that could be served from the same computer running the database itself, thereby allowing you to connect remotely.

Some of the ones we've used include:

- phpMyAdmin
- MySQL Workbench
- Navicat (commercial)

In our examples we will be using the MySQL command line, not because it is superior, but simply because it's ubiquitous on any system with MySQL, so you've already got it and have been using it in this book.

For more heavy-duty database design, the command line is probably not as powerful as a well-designed GUI would be. Grab a copy of MySQL Workbench at least and give it a whirl.

Troubleshooting Database Issues

When working with ODBC database connections and Asterisk, it is important to remember that the ODBC connection abstracts some of the information passed between Asterisk and the database. In cases where things are not working as expected, you may need to enable logging on your database platform to see what Asterisk is sending to the database (e.g., which `SELECT`, `INSERT`, or `UPDATE` statements are being triggered from Asterisk), what the database is seeing, and why the database may be rejecting the statements.

For example, one of the most common problems found with ODBC database integration is an incorrectly defined table or a missing column that Asterisk expects to exist. While great strides have been made in the form of adaptive modules, not all parts of Asterisk are adaptive. In the case of ODBC voicemail storage, you may have missed a column such as `flag`, which is a new column not found in versions of Asterisk prior to 11.[1] As noted, in order to debug why your data is not being written to the database as expected, you should enable statement logging on the database side, and then determine what statement is being executed and why the database is rejecting it.

SQL Injection

Security is always a consideration when you are building networked applications, and database security is no exception.

In the case of Asterisk, you have to think about what input you are accepting from users (typically what they are able to submit to the dialplan), and work to sanitize that input to ensure you are only allowing characters that are valid to your application. As an example, a typical telephone call would only allow digits as input (and possibly the `*` and `#` characters), so there would be no reason to accept any other characters. Bear in mind that the SIP protocol allows more than just numbers as part of an address, so don't assume that somebody attempting to compromise your system is limited to just digits.

A little extra time spent sanitizing your allowed input will improve the security of your application.

Powering Your Dialplan with func_odbc

The `func_odbc` dialplan function module allows you to define and use relatively simple functions in your dialplan that will retrieve information from databases as calls are being processed. There are all kinds of ways in which this might be used, such as

1 This was actually an issue one of the authors had while working on this book, and he found the `flag` column by looking at the statement logging during testing.

managing users or allowing the sharing of dynamic information within a clustered set of Asterisk machines. We won't claim that this will make designing and writing dialplan code easier, but we will promise that it will allow you to add a whole new level of power to your dialplans, especially if you are comfortable working with databases. We don't know anybody in the Asterisk community who does not love `func_odbc`.

The way `func_odbc` works is by allowing you to define SQL queries, to which you assign function names. The *func_odbc.conf* file is where you specify the relationships between the functions you create and the SQL statements you wish them to perform. You use the named functions you have created in your dialplan to retrieve and update values in the database.

In order to get you into the right frame of mind for what follows, we want you to picture a Dagwood sandwich.[2]

Can you relay the total experience of such a thing by showing someone a picture of a tomato, or by waving a slice of cheese about? Hardly. That is the conundrum we faced when trying to give useful examples of why `func_odbc` is so powerful. So, we decided to build the whole sandwich for you. It's quite a mouthful, but after a few bites of this, peanut butter and jelly is never going to be the same.

ODBC Configuration File Relationships

Several files must all line up in order for Asterisk to be able to use ODBC from the dialplan. Figure 15-1 attempts to convey this visually. You will probably find this diagram more helpful once you have worked through the examples in the following sections.

2 And if you don't know what a Dagwood is, that's what Wikipedia is for. I am not that old.

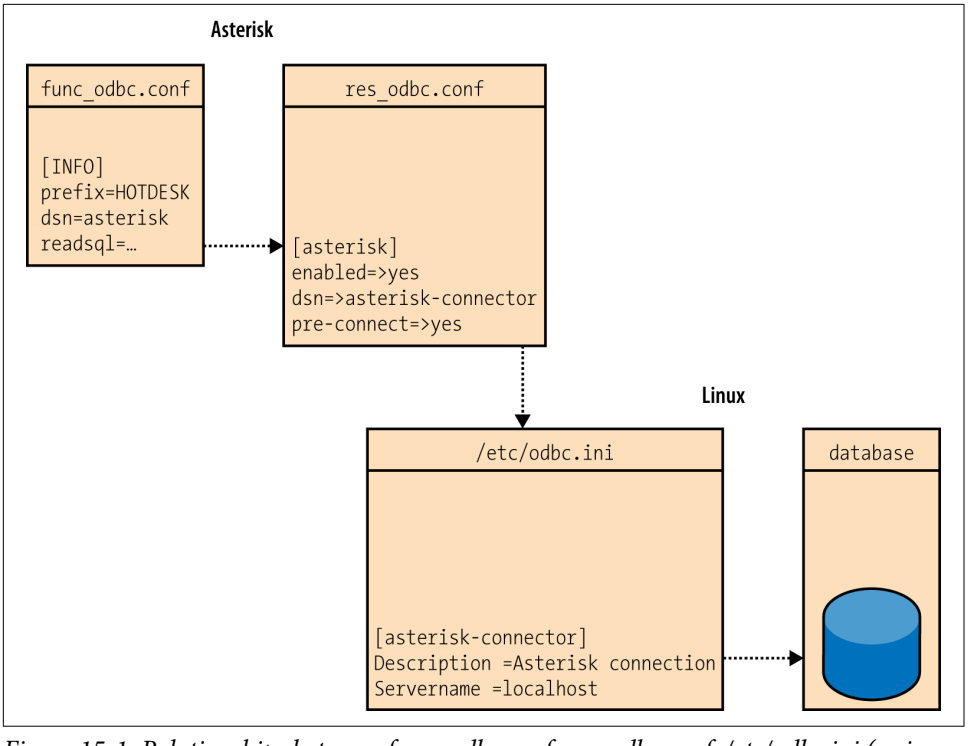

Figure 15-1. Relationships between func_odbc.conf, res_odbc.conf, /etc/odbc.ini (unixODBC), and the database connection

A Gentle Introduction to func_odbc

Before we dive into `func_odbc`, we feel a wee bit of history is in order.

The very first use of `func_odbc`, which occurred while its author was still writing it, is also a good introduction to its use. A customer of one of the module's authors noted that some people calling into his switch had figured out a way to make free calls with his system. While his eventual intent was to change his dialplan to avoid those problems, he needed to blacklist certain caller IDs in the meantime, and the database he wanted to use for this was a Microsoft SQL Server database.

With a few exceptions, this is the actual dialplan:

```
[span3pri]
exten => _50054XX,1,NoOp()
   same => n,Set(CDR(accountcode)=pricall)
   ; Does this callerID appear in the database?
   same => n,GotoIf($[${ODBC_ANIBLOCK(${CALLERID(number)})}]?busy)
   same => n(dial),Dial(DAHDI/G1/${EXTEN})
   same => n(busy),Busy(10) ; Yes, you are on the blacklist.
   same => n,Hangup
```

This dialplan, in a nutshell, passes all calls to another system for routing purposes, except those calls whose caller IDs are in a blacklist. The calls coming into this system used a block of 100 seven-digit DIDs. You will note a dialplan function is being used that you won't find listed in any of the functions that ship with Asterisk: `ODBC_ANIBLOCK()`. This function was instead defined in another configuration file, *func_odbc.conf*:

```
[ANIBLOCK]
dsn=telesys
readsql=SELECT IF(COUNT(1)>0, 1, 0) FROM Aniblock WHERE NUMBER='${ARG1}'
```

So, your `ODBC_ANIBLOCK()`[3] function connects to a data source in *res_odbc.conf* named `telesys` and selects a count of records that have the `NUMBER` specified by the argument, which is (referring to the preceding dialplan) the caller ID. Nominally, this function should return either a `1` (indicating the caller ID exists in the `Aniblock` table) or a `0` (if it does not). This value also evaluates directly to true or false, which means we don't need to use an expression in our dialplan to complicate the logic.

And that, in a nutshell, is what `func_odbc` is all about: writing custom dialplan functions that return a result from a database. Next up, a more detailed example of how one might use `func_odbc`.

Getting Funky with func_odbc: Hot-Desking

OK, back to the Dagwood sandwich we promised.

We believe the value of `func_odbc` will become very clear to you if you work through the following example, which will produce a new feature on your Asterisk system that depends heavily on `func_odbc`.

Picture a small company with a sales force of five people who have to share two desks. This is not as cruel as it seems, because these folks spend most of their time on the road, and they are each only in the office for at most one day each week.

Still, when they do get into the office, they'd like the system to know which desk they are sitting at, so that their calls can be directed there. Also, the boss wants to be able to track when they are in the office and control calling privileges from those phones when no one is there.

This need is typically solved by what is called a *hot-desking* feature. We have built one for you in order to show you the power of `func_odbc`.

3 We're using the `IF()` SQL function to make sure we return a value of `0` or `1`. This works on MySQL 5.1 or later. If it does not work on your SQL installation, you could also check the returned result in the dialplan using the `IF()` function there.

Let's start with the easy stuff, and create two new phone credentials in our database.

First, the endpoints table:

```
MySQL> INSERT INTO asterisk.ps_endpoints (id,transport,aors,auth,context,disallow,allow, \
direct_media,callerid)

VALUES
('HOTDESK_1','transport-tls','HOTDESK_1','HOTDESK_1','hotdesk','all','ulaw','no', \
'HOTDESK_1'),
('HOTDESK_2','transport-tls','HOTDESK_2','HOTDESK_2','hotdesk','all','ulaw','no', \
'HOTDESK_2');
```

Then, the `auths`:

```
MySQL> INSERT INTO asterisk.ps_auths (id,auth_type,password,username)

VALUES
('HOTDESK_1','userpass','notsohot1','HOTDESK_1'),
('HOTDESK_2','userpass','notsohot2','HOTDESK_2');
```

Finally, the `aors`:

```
MySQL> INSERT INTO asterisk.ps_aors
(id,max_contacts)

VALUES
('HOTDESK_1',1),
('HOTDESK_2',1);
```

Notice that we've told these two endpoints to enter the dialplan at a context named [`hotdesk`]. We'll define that shortly.

That's all for our endpoint configuration. We've got a few slices of bread, which is hardly a sandwich yet.

Now let's get the custom database built that we're going to use for this.

Connect to your MySQL console as `root`:

```
$ mysql -u root -p
```

First we want a new schema to put all this in. It's technically possible to put this in the `asterisk` schema, but we prefer to leave that schema alone, reserved only for whatever Asterisk's Alembic scripts do with it during upgrades.

```
MySQL> CREATE SCHEMA pbx;

MySQL> GRANT SELECT,INSERT,UPDATE,DELETE,EXECUTE,SHOW VIEW ON pbx.* TO 'asterisk'@'::1';

MySQL> GRANT SELECT,INSERT,UPDATE,DELETE,EXECUTE,SHOW VIEW ON pbx.* TO \
'asterisk'@'127.0.0.1';

MySQL> GRANT SELECT,INSERT,UPDATE,DELETE,EXECUTE,SHOW VIEW ON pbx.* TO \
'asterisk'@'localhost';

MySQL> GRANT SELECT,INSERT,UPDATE,DELETE,EXECUTE,SHOW VIEW ON pbx.* TO \
'asterisk'@'localhost.localdomain';
```

```
MySQL> FLUSH PRIVILEGES;
```

Then create the table with the following bit of SQL:

```
CREATE TABLE pbx.ast_hotdesk
(
  id serial NOT NULL,
  extension text,
  first_name text,
  last_name text,
  cid_name text,
  cid_number varchar(10),
  pin int,
  status bool DEFAULT false,
  endpoint text,
  CONSTRAINT ast_hotdesk_id_pk PRIMARY KEY (id)
);
```

After that, populate the database with the following information (some of the values that you see actually will change only after the dialplan work is done, but we include it here by way of example).

At the MySQL console, run the following command:

```
MySQL> INSERT INTO pbx.ast_hotdesk
(extension, first_name, last_name, cid_name, cid_number, pin, status)

VALUES
('1101','Herb','Tarlek','WKRP','1101','110111',0)
('1102','Al','Bundy','Garys','1102','110222',0),
('1103','Willy','Loman','','1103','110333',0),
('1104','Jerry','Lundegaard','Gustafson','1104','110444',0),
('1105','Moira','Brown','Craterside','1105','110555',0);
```

Repeat these commands, changing the `VALUES` as needed, for all entries you wish to have in the database.[4] After you've input your sample data, you can view the data in the `ast_hotdesk` table by running a simple `SELECT` statement from the database console:

```
MySQL> SELECT * FROM pbx.ast_hotdesk;
```

Which might give you something like the following output:

```
+--+---------+----------+----------+----------+----------+------+------+--------+
|id|extension|first_name|last_name |cid_name  |cid_number|pin   |status|endpoint|
+--+---------+----------+----------+----------+----------+------+------+--------+
| 1|1101     |Herb      |Tarlek    |WKRP      |1101      |110111|     0|NULL    |
| 2|1102     |Al        |Bundy     |Garys     |1102      |110222|     0|NULL    |
| 3|1103     |Willy     |Loman     |          |1103      |110333|     0|NULL    |
| 4|1104     |Jerry     |Lundegaard|Gustafson |1104      |110444|     0|NULL    |
| 5|1105     |Moira     |Brown     |Craterside|1105      |110555|     0|NULL    |
+--+---------+----------+----------+----------+----------+------+------+--------+
```

4 Note that in the first example user, we are assigning a status of 1 and a location, whereas for the second example user, we are not defining a value for these fields.

We've got the condiments now, so let's get to our dialplan. This is where the magic is going to happen.

Somewhere in *extensions.conf* we are going to create the `[hotdesk]` context. To start, let's define a pattern-match extension that will allow the users to log in:

```
[hotdesk]
include => sets

exten => _*99110[1-5],1,Noop(Hotdesk login)
  same => n,Set(HotExten=${EXTEN:3}) ; strip off the leading *99
  same => n,Noop(Hotdesk Extension ${HotExten} is changing status) ; for the log
  same => n,Set(${HotExten}_STATUS=${HOTDESK_INFO(status,${HotExten})})
  same => n,Set(${HotExten}_PIN=${HOTDESK_INFO(pin,${HotExten})})
  same => n,Noop(${HotExten}_PIN is now ${${HotExten}_PIN})
  same => n,Noop(${HotExten}_STATUS is ${${HotExten}_STATUS})})
```

We're not done writing this extension yet, but we need to digress for a few pages to discuss where we're at so far.

When a sales agent sits down at a desk, they log in by dialing *99 plus their extension number. In this case we have allowed the 1101 through 1105 extensions to log in with our pattern match of `_99110[1-5]`. You could just as easily make this less restrictive by using `_9911XX` (allowing 1100 through 1199). This extension uses `func_odbc` to perform a lookup with the `HOTDESK_INFO()` dialplan function. This custom function (which we will define in the *func_odbc.conf* file) performs an SQL statement and returns whatever is retrieved from the database.

So, let's create the */etc/asterisk/func_odbc.conf* file, and within that define the new function `HOTDESK_INFO()`:

```
$ sudo -u asterisk vim /etc/asterisk/func_odbc.conf

[INFO]
prefix=HOTDESK
dsn=asterisk
synopsis=Select value of field in ARG1, where 'extension' matches ARG2
description=Allow dialplan to extract data from any field in pbx.ast_hotdesk table.
readsql=SELECT ${ARG1} FROM pbx.ast_hotdesk WHERE extension = '${ARG2}'
```

That's a lot of stuff in just a few lines. Let's quickly cover them before we move on.

You should be able to reload your dialplan (`dialplan reload`) and `func_odbc` (`module reload func_odbc.so`), and test the dialplan out thus far (dial 991101 from one of the sets you've assigned to this context). Make sure your console verbosity is set to at least 3 (`*CLI> core set verbose 3`), as you will only be able to see this dialplan working by the console (a call to this dialplan will return a fast busy even if it runs successfully). For the rest of this section, we strongly recommend you test everything after each change. If you don't, you'll have a whale of a time trying to find the bugs. It's critical that you code with a phone registered and the Asterisk console open, so you can reload and test changes within seconds of writing them.

First of all, the `prefix` is optional (default prefix is `'ODBC'`). This means that if you don't define a `prefix`, Asterisk adds `'ODBC'` to the function name (in this case, `INFO`), which means this function would become `ODBC_INFO()`. This is not very descriptive of what the function is doing, so it can be helpful to assign a prefix that helps to relate your ODBC functions to the tasks they are performing. We chose `'HOTDESK'`, which means that this custom function will be named `HOTDESK_INFO()` in the dialplan.

The reason why `prefix` is separate is that the author of the module wanted to reduce possible collisions with existing dialplan functions. The intent of `prefix` was to allow multiple copies of the same function, connected to different databases, for multitenant Asterisk systems. We as authors have been a bit more liberal in our use of `prefix` than the developer originally intended.

The `dsn` attribute tells Asterisk which connection to use from *res_odbc.conf*. Since several database connections could be configured in *res_odbc.conf*, we specify which one to use here. In Figure 15-1, we show the relationship between the various file configurations and how they reference down the chain to connect to the database.

The *func_odbc.conf.sample* file in the Asterisk source contains additional information about how to handle multiple databases and control the reading and writing of information to different DSN connections. Specifically, the `readhandle`, `writehandle`, `readsql`, and `writesql` arguments will provide you with great flexibility for database integration and control.

Finally, we define our SQL statement with the `readsql` attribute. Dialplan functions can be called with two different formats: one for retrieving information, and one for setting information. The `readsql` attribute is used when we call the `HOTDESK_INFO()` function with the retrieve format (we could execute a separate SQL statement with

the writesql attribute; we'll discuss the format for that attribute a little bit later in this chapter).

Reading values from this function would take this format in the dialplan:

```
exten => s,n,Set(RETURNED_VALUE=${HOTDESK_INFO(status,1101)})
```

This would return the value located in the database within the status column where the extension column equals 1101. The status and 1101 we pass to the HOTDESK_INFO() function are then placed into the SQL statement we assigned to the readsql attribute, available as ${ARG1} and ${ARG2}, respectively. If we had passed a third option, this would have been available as ${ARG3}.

After the SQL statement is executed, the value returned (if any) is assigned to the RETURNED_VALUE channel variable.

Using the ARRAY() Function

In our example, we are utilizing two separate database calls and assigning those values to a pair of channel variables: ${HotdeskExtension}_STATUS and ${HotdeskExtension}_PIN. This was done to simplify the example. We're going to shorten the names of the variables here because the printed format can't handle such long lines, so in the following examples, you'll see "HE" in place of "HotdeskExtension." If you're going to code this example, please replace HE with HotdeskExtension:

```
same => n,Set(${HE}_STATUS=${HOTDESK_INFO(status,${HE})})
same => n,Set(${HE}_PIN=${HOTDESK_INFO(pin,${HE})})
```

As an alternative, we could have returned multiple columns and saved them to separate variables utilizing the ARRAY() dialplan function. If we had defined our SQL statement in an *func_odbc.conf* function like so:

```
readsql=SELECT pin,status FROM ast_hotdesk WHERE extension = '${HE}'
```

we could have used the ARRAY() function to save each column of information for the row to its own variable with a single call to the database (note that we're using an example function called HOTDESK_INFO(), which we haven't created):

```
same => n,Set(ARRAY(${HE}_PIN,${HE}_STATUS)=${HOTDESK_INFO(${HE})})
```

Using ARRAY() is handy any time you might get comma-separated values back and want to assign the values to separate variables, such as with CURL(). However, it can also make your code more complicated to read, debug, and maintain.

So, in the first two lines of the following block of code, we are passing the value status and the value contained in the ${HotdeskExtension} variable (e.g., 1101) to the HOTDESK_INFO() function. The two values are then replaced in the SQL statement

with ${ARG1} and ${ARG2}, respectively, and the SQL statement is executed. Finally, the value returned is assigned to the ${HotdeskExtension}_STATUS channel variable.

Let's finish writing the pattern-match extension now:

```
exten => _*99110[1-5],1,Noop(Hotdesk login)
  same => n,Set(HotdeskExtension=${EXTEN:3}) ; strip off the leading *99
  same => n,Noop(Hotdesk Extension ${HotdeskExtension} is changing status) ; for the log
  same => n,Set(${HotdeskExtension}_STATUS=${HOTDESK_INFO(status,${HotdeskExtension})})
  same => n,Set(${HotdeskExtension}_PIN=${HOTDESK_INFO(pin,${HotdeskExtension})})
  same => n,Noop(${HotdeskExtension}_PIN is now ${${HotdeskExtension}_PIN})
  same => n,Noop(${HotdeskExtension}_STATUS is ${${HotdeskExtension}_STATUS})})
  same => n,GotoIf($["${${HotdeskExtension}_PIN}" = ""]?invalid_user)
  same => n,GotoIf($[${ODBCROWS} < 0]?invalid_user)
  same => n,GotoIf($[${${HotdeskExtension}_STATUS} = 1]?logout:login,1)
```

We'll be writing some labels to handle `invalid_user` and `logout` a bit later, so don't worry if it seems something is missing.

You may have noticed that in some of the `Goto/GotoIf` examples, there might be a `,1` in the directive. This might seem confusing unless you recall that the target only needs the difference between the current *`context`*,*`extension`*,*`priority/label`*. So, if you send something to a label, such as `logout`, that is in the same extension, you don't need to specify the context and extension, whereas if you are sending the call to the extension named `login` (still in the same context), you need to specify that you wish the call to go to label/priority `1`. In the previous example, we could write our directive as follows:

```
... = 1] ? hotdesk,${EXTEN},logout : hotdesk,login,1
           ^same    ^same   ^diff    ^same   ^diff ^diff
```

In other words, **`true`** goes to context `[hotdesk]`, extension `99110[1-5]`, label `logout`; and **`false`** goes to context `[hotdesk]`, extension `login`, and label/priority `1`.

We only wrote what's different.

If you want, for clarity, you can always write *`context`*,*`extension`*,*`priority`* for all your directives. It's your call.

After assigning the value of the `status` column to the `${HotdeskExtension}_STATUS` variable (if the user identifies themself as extension `1101`, the variable name will be `1101_STATUS`), we check if we've received a value back from the database using the `${ODBCROWS}` channel variable.

The last row in the block checks the status of the phone and, if the agent is currently logged in, logs them off. If the agent is not already logged in, it will go to the `login` extension.

At the `login` extension the dialplan runs some initial checks to verify the PIN code entered by the agent. (Additionally, we've used the `FILTER()` function to make sure only numbers were entered to help avoid some SQL injection issues.) We allow them three tries to enter the correct PIN, and if all tries are invalid we'll hang up:

```
exten => login,1,NoOp() ; set initial counter values
   same => n,Set(PIN_TRIES=1)     ; pin tries counter
   same => n,Set(MAX_PIN_TRIES=3) ; set max number of login attempts
   same => n,Playback(silence/1)  ; play back some silence so first prompt is
                                  ; not cut off
   same => n(get_pin),NoOp()
   same => n,Set(PIN_TRIES=$[${PIN_TRIES} + 1])   ; increase pin try counter
   same => n,Read(PIN_ENTERED,enter-password,${LEN(${${HotdeskExtension}_PIN})})
   same => n,Set(PIN_ENTERED=${FILTER(0-9,${PIN_ENTERED})})
   same => n,GotoIf($["${PIN_ENTERED}" = "${${HotdeskExtension}_PIN}"]?valid:invalid)
   same => n,Hangup()

   same => n(invalid),Playback(vm-invalidpassword)
   same => n,GotoIf($[${PIN_TRIES} <= ${MAX_PIN_TRIES}]?get_pin)
   same => n,Playback(goodbye)
   same => n,Hangup()

   same => n(valid),Noop(Valid PIN)
```

If the PIN entered matches, we continue with the login process through the `(valid)` label. First we utilize the `CHANNEL` variable to figure out which phone device the agent is calling from. The `CHANNEL` variable is usually populated with something similar to `PJSIP/HOTDESK_1-ab4034c`, so we make use of the `CUT()` function to first strip off the `PJSIP/` component of the string. We then strip off the `-ab4034c` part of the string, and what remains is what we want (`HOTDESK_1`):[5]

```
   same => n(valid),Noop(Valid PIN)
; CUT off the channel technology and assign it to the LOCATION variable
   same => n,Set(LOCATION=${CUT(CHANNEL,/,2)})
; CUT off the unique identifier and save the remainder to the LOCATION variable
   same => n,Set(LOCATION=${CUT(LOCATION,-,1)})
; we'll come back to this shortly
```

We're going to create and use some more functions in the *func_odbc.conf* file: `HOTDESK_CHECK_SET()`, which will determine if other users are already assigned to this phone; `HOTDESK_STATUS()`, which will assign the phone to this agent; and `HOTDESK_CLEAR_SET()`, which will clear any other users currently assigned to this phone (who perhaps forgot to log out).

In our *func_odbc.conf* file we'll need to create the following functions:

```
; func_odbc.conf
[CHECK_SET]
prefix=HOTDESK
```

5 Yes, you can nest functions within functions, and so do this all on one line. We didn't do so as it's more difficult to debug, and doesn't affect performance.

```
dsn=asterisk
synopsis=Check if this set is already assigned to somebody.
readsql=SELECT COUNT(status) FROM pbx.ast_hotdesk WHERE status = '1'
readsql+= AND endpoint = '${ARG1}'

[STATUS]
prefix=HOTDESK
dsn=asterisk
synopsis=Assign hotdesk extension to this endpoint/set.
writesql=UPDATE pbx.ast_hotdesk SET status = '${SQL_ESC(${VAL1})}',
writesql+= endpoint = '${SQL_ESC(${VAL2})}'
writesql+= WHERE extension = '${SQL_ESC(${ARG1})}'

[CLEAR_SET]
prefix=HOTDESK
dsn=asterisk
synopsis=Clear all instances of this endpoint
writesql=  UPDATE pbx.ast_hotdesk SET status=0,endpoint=NULL
writesql+= WHERE endpoint='${SQL_ESC(${VAL1})}'
```

Due to line-length limitations in the book, we've broken the `readsql` and `writesql` commands into multiple lines using the `+=` syntax, which tells Asterisk to append the contents after `readsql+=` to the most recently defined `readsql=` value (or `writesql` and `writesql+`). The usage of `+=` is applicable not only to the `readsql` option, but can be used in other places in other *.conf* files within Asterisk.

In our dialplan, we'll need to call the function we just created, and pass call flow to the `forcelogout` label if somebody is already logged into this set:

```
  same => n(valid),Noop(Valid PIN)
  same => n,Set(LOCATION=${CUT(CHANNEL,/,2)})
  same => n,Set(LOCATION=${CUT(LOCATION,-,1)})
; We'll come back to this shortly ; you can remove this comment/line
  same => n(checkset),Set(SET_USED=${HOTDESK_CHECK_SET(${LOCATION})})
  same => n,GotoIf($[${SET_USED} > 0]?forcelogout)

; Set status for agent  to '1' and update the location/endpoint
  same => n(set_login_status),Set(HOTDESK_STATUS(${HotdeskExtension})=1,${LOCATION})
  same => n,Noop(ODBCROWS is ${ODBCROWS})
  same => n,GotoIf($[${ODBCROWS} < 1]?error,1)
  same => n,Playback(agent-loginok)
  same => n,Hangup()

  same => n(forcelogout),NoOp()
; set all currently logged-in users on this device to logged-out status
  same => n,Set(HOTDESK_CLEAR_SET()=${LOCATION})
  same => n,Goto(checkset)      ; return to logging in
```

There are some potentially new concepts we've just introduced in the examples. Specifically, the syntax in the `HOTDESK_STATUS()` function has a few new tricks you might have noticed. We now have both `${VALx}` and `${ARGx}` variables in our SQL statement.

We've wrapped the `${VALx}` and `${ARGx}` values in the `SQL_ESC()` function as well, which will escape characters such as backticks that could be used in an SQL injection attack.

These contain the information we pass to the function from the dialplan. In this case, we have two `VAL` variables and a single `ARG` variable that were set from the dialplan via this statement:

```
same => n(set_login_status),Set(HOTDESK_STATUS(${HotdeskExtension})=1,${LOCATION})
```

Notice the syntax is slightly different from that of the read-style function. This signals to Asterisk that you want to perform a write (this is the same structural syntax as that used for other dialplan functions).

We are including the value of the `${HotdeskExtension}` variable in our call to the `HOTDESK_STATUS()` function (which then becomes the `${ARG1}` variable for that function in *func_odbc.conf*). However, we are also passing two values, `'1'` and `${LOCATION}`. These will be associated in the function by the `${VAL1}` and `${VAL2}` variables, respectively.

Using SQL Directly in Your Dialplan

Some people would prefer to write their SQL statements in the dialplan directly, as opposed to crafting a custom function for each type of database transaction they might want to perform.

In theory, you could create just one function in *func_odbc.conf* like this:

```
[SQL]
prefix=GENERIC
dsn=asterisk
readsql=${SQL_ESC(${ARG1})}
writesql=${SQL_ESC(${VALUE})} ; Whole value, un-parsed
```

Then, in your dialplan you could write pretty much any sort of SQL you wanted (provided the ODBC connector could handle it, which has nothing to do with Asterisk). That one function above would then submit whatever string you specified directly to the ODBC connection to your database.[6]

Some would argue this makes for more confusion in your dialplan; others will insist that the benefit of having a much simpler *func_odbc.conf* file is worth it:

```
same => n,Set(result=${GENERIC_SQL(SELECT col FROM table WHERE ...)})
same => n,Verbose(1,${result})
```

6 It could also pose a needless security risk.

```
same => n,Set(GENERIC_SQL()=UPDATE table SET field="VAL" WHERE ...)
same => n,Verbose(1,ODBC_RESULT is ${OBDBC_RESULT})
```

We believe it's generally better to build specific functions in *func_odbc.conf* to handle queries from your dialplan; however, there's no denying the temptation to use one function to handle all SQL queries.

Multirow Functionality with func_odbc

Asterisk has a *multirow* mode that allows it to handle multiple rows of data returned from the database. For example, if we were to create a dialplan function in *func_odbc.conf* that returned all available extensions, we would need to enable multi-row mode for the function. This would cause the function to work a little differently, returning an ID number that could then be passed to the `ODBC_FETCH()` function to return each row in turn.

A simple example follows. Suppose we have the following *func_odbc.conf*:

```
[AVAILABLE_EXTENS]
prefix=HOTDESK
dsn=asterisk
mode=multirow
readsql=SELECT extension FROM ast_hotdesk WHERE status = '${ARG1}'
```

and a dialplan in *extensions.conf* that looks something like this:

```
exten => *9997,1,Noop(multirow)
 same => n,Set(ODBC_ID=${HOTDESK_AVAILABLE_EXTENS()})
 same => n,GotoIf($[${ODBCROWS} < 1]?no_rows)
 same => n,Answer()
 same => n,Set(COUNTER=1)
 same => n,While($[${COUNTER} <= ${ODBCROWS}])
   same => n,Set(AVAIL_EXTEN_${COUNTER}=${ODBC_FETCH(${ODBC_ID})})
   same => n,SayDigits(${AVAIL_EXTEN_${COUNTER}})
   same => n,Wait(0.2) ; Pause between speaking
   same => n,Set(COUNTER=$[${COUNTER} + 1])
 same => n,EndWhile()
 same => n(norows),ODBCFinish()
 same => n,Hangup()
```

Note that unless you have multiple endpoints to log in, this will never return more than one extension in your lab because only one device will be logged in at any time. You can add some dummy data to the table just to see how this works:

```
MySQL> UPDATE pbx.ast_hotdesk
       SET status='1',endpoint='HOTDESK_2'
       WHERE id='3'
       ;
MySQL> UPDATE pbx.ast_hotdesk
       SET status='1',endpoint='HOTDESK_3'
       WHERE id='5'
       ;
```

The ODBC_FETCH() function will essentially treat the information as a stack, and each call to it with the passed ODBC_ID will pop the next row of information off the stack. We also have the option of using the ODBC_FETCH_STATUS channel variable, which is set once the ODBC_FETCH() function (which returns SUCCESS if additional rows are available or FAILURE if no additional rows are available) is called. This permits us to write a dialplan like the following, which does not use a counter, but still loops through the data. This may be useful if we're looking for something specific and don't need to look at all the data. Once we're done, the ODBCFinish() dialplan application should be called to clean up any remaining data.

Here's another *extensions.conf* example:

```
[multirow_example_2]
exten => start,1,Verbose(1,Looping example with break)
   same => n,Set(ODBC_ID=${GET_ALL_AVAIL_EXTENS(1)})
   same => n(loop_start),NoOp()
   same => n,Set(ROW_RESULT=${ODBC_FETCH(${ODBC_ID})})
   same => n,GotoIf($["${ODBC_FETCH_STATUS}" = "FAILURE"]?cleanup,1)
   same => n,GotoIf($["${ROW_RESULT}" = "1104"]?good_exten,1)
   same => n,Goto(loop_start)

exten => cleanup,1,Verbose(1,Cleaning up after all iterations)
   same => n,Verbose(1,We did not find the extension we wanted)
   same => n,ODBCFinish(${ODBC_ID})
   same => n,Hangup()

exten => good_exten,1,Verbose(1,Extension we want is available)
   same => n,ODBCFinish(${ODBC_ID})
   same => n,Verbose(1,Perform some action we wanted)
   same => n,Hangup()
```

OK, we've digressed a bit. Let's wrap up a few parts of the agent components that we haven't handled yet.

In the _*99110[1-5] extension, we need the following labels:

```
same => n,GotoIf($[${${HotdeskExtension}_STATUS} = 1]?logout:login,1)

same => n(invalid_user),Noop(Hot Desk extension ${HotdeskExtension} does not exist)
same => n,Playback(silence/2&login-fail)
same => n,Hangup()

same => n(logout),Noop()
same => n,Set(HOTDESK_STATUS(${HotdeskExtension})=0,) ; Note VAL2 is empty
same => n,GotoIf($[${ODBCROWS} < 1]?error,1)
same => n,Playback(silence/1&agent-loggedoff)
same => n,Hangup()
```

We also include the hotdesk_outbound context, which will handle our outgoing calls after we have logged the agent into the system:

```
include => hotdesk_outbound ; this line can go anywhere in the [hotdesk] context
```

The [hotdesk_outbound] context utilizes many of the same principles already discussed. This context uses a pattern match to catch any numbers dialed from the hotdesk phones. We first set our LOCATION variable using the CHANNEL variable, then determine which extension (agent) is logged into the system and assign that value to the WHO variable. If this variable is NULL, we reject the outgoing call. If it is not NULL, then we get the agent information using the HOTDESK_INFO() function and assign it to several CHANNEL variables.

```
include => hotdesk_outbound

; put this code right below your [hotdesk] context
[hotdesk_outbound]
exten => _NXXXXXX.,1,NoOp()
 same => n,Set(LOCATION=${CUT(CHANNEL,/,2)})
 same => n,Set(LOCATION=${CUT(LOCATION,-,1)})
 same => n(checkset),Set(VALID_AGENT=${HOTDESK_CHECK_SET(${LOCATION})})
 same => n,Noop(VALID_AGENT is ${VALID_AGENT})
 same => n,Set(${CALLERID(name)}=${HOTDESK_INFO(cid_name,${VALID_AGENT})})
 same => n,Set(${CALLERID(num)}=${HOTDESK_INFO(cid_number,${VALID_AGENT})})
 same => n,GotoIf($[${VALID_AGENT} = 0]?notallowed) ; Nobody logged in--calls not allowed
 same => n,Dial(${LOCAL}/${EXTEN}) ; See the Outside Connectivity chapter
 same => n,Hangup()

 same => n(notallowed),Playback(sorry-cant-let-you-do-that2)
 same => n,Hangup()
```

If you are not logged in, the call will fail with a message. If you are logged in, the call will be passed to the Dial() application (which might also fail if you don't have a carrier configured, but that's something covered in earlier chapters, so we're going to leave it as this for this section).

There's one last bit of dialplan required. We have built this complex environment that lets our agents log in and out, but there isn't actually any way of calling them!

We're going to fix that now, by doing four things:

1. We're going to include the [sets] context in the [hotdesk] context, so that our agents can use the other parts of our dialplan.
2. We're going to give our agents mailboxes.
3. We're going to create a new subroutine that will check the hotdesk for an agent, and a) ring them if they're there, or b) fire the call off to voicemail if they're not.
4. We're going to build dialplan in the [sets] context so that everyone can call our agents.

Let's get the mailboxes out of the way first:

```
MySQL> insert into `asterisk`.`voicemail`
(mailbox,fullname,context,password)
VALUES
```

```
('1101','Herb Tarlek','default','110111'),
('1102','Al Bundy','default','110222'),
('1103','Willy Loman','default','110333'),
('1104','Jerry Lundegaard','default','110444'),
('1105','Moira Brown','default','110555');
```

All the rest of the work is in *extensions.conf*:

Way down at the bottom, let's craft a subroutine that'll handle things for us:

```
[subDialHotdeskUser]
exten => _[a-zA-Z0-9].,1,Noop(Call Hotdesk)
 same => n,Set(HOTDESK_ENDPOINT=${HOTDESK_INFO(endpoint,${EXTEN})}) ; Get assigned device
 same => n,GotoIf($["${HOTDESK_ENDPOINT}" = ""]?voicemail) ; if blank, send to voicemail
 same => n(ringhotdesk),Dial(PJSIP/${HOTDESK_ENDPOINT},${ARG1})
 same => n(voicemail),Voicemail(${EXTEN})
 same => n,Hangup()
```

And somewhere far closer to the top, we'll add our hotdesk users to the section of dialplan where our other users live:

```
exten => 110,1,Dial(${UserA_DeskPhone}&${UserA_SoftPhone}&${UserB_SoftPhone})

exten => 1101,1,GoSub(subDialHotdeskUser,${EXTEN},1(12))
exten => 1102,1,GoSub(subDialHotdeskUser,${EXTEN},1(12))
exten => 1103,1,GoSub(subDialHotdeskUser,${EXTEN},1(12))
exten => 1104,1,GoSub(subDialHotdeskUser,${EXTEN},1(12))
exten => 1105,1,GoSub(subDialHotdeskUser,${EXTEN},1(12))

exten => 200,1,Answer()
     same => n,Playback(hello-world)
     same => n,Hangup()
```

And finally, back in our `[hotdesk]` context, we're going to allow our agents to use the rest of the phone system:

```
[hotdesk]

include => sets

exten => _*99110[1-5],1,Noop(Hotdesk login)
```

Try a few scenarios:

1. Call from an agent internally.
2. Call from a normal user to a logged-in agent.
3. Call from a normal user to an unavailable agent.

Marvel at this technological terror you've constructed.

Now that we've implemented a fairly complex feature in the dialplan, using `func_odbc` to retrieve and store data in a remote relational database, you can see that with a handful of fairly simple functions in the *func_odbc.conf* file and a couple of tables in a database, you can create some powerful telephony applications.

OK, let's move on to the Asterisk Realtime Architecture, which has in many cases been made obsolete by ODBC, but can still be useful.

Using Realtime

The Asterisk Realtime Architecture (ARA) allows you to store all the parameters normally stored in your Asterisk configuration files (commonly located in */etc/asterisk*) in a database. There are two types of realtime: *static* and *dynamic*.

The static version is similar to the traditional method of reading a configuration file (information is only loaded when triggered from the CLI), except that the data is read from the database instead.[7]

The Dynamic Realtime method, which loads and updates the information as it is used by the live system, is commonly used for things such as SIP (or IAX2, etc.) user and peer objects, as well as voicemail boxes.

Making changes to static information requires a reload, just as if you had changed a text file on the system, but dynamic information is polled by Asterisk as needed, so no reload is required when changes are made to this data. Realtime is configured in the *extconfig.conf* file located in the */etc/asterisk* directory. This file tells Asterisk what to load from the database and where to load it from, allowing certain files to be loaded from the database and other files to be loaded from the standard configuration files.

7 Yes, calling this "realtime" is somewhat misleading, as updates to the data will not affect anything happening in real time (until a reload of the relevant module is performed).

Another (arguably older) way to store Asterisk configuration was through an external script, which would interact with a database and generate the appropriate flat files (or *.conf* files), and then reload the appropriate module once the new file was written. There is an advantage to this (if the database goes down, your system will continue to function; the script will simply not update any files until connectivity to the database is restored), but it also has disadvantages. One major disadvantage is that any changes you make to a user will not be available until you run the update script. This is probably not a big issue on small systems, but on large systems, waiting for changes to take effect can cause issues, such as pausing a live call while a large file is loaded and parsed.

You can relieve some of this by utilizing a replicated database system. Asterisk provides the ability to fail over to another database system. This way, you can cluster the database backend utilizing a master-master relationship (for PostgreSQL, pgcluster, or Postgre-R;[8] for MySQL it's native[9]), or a master-slave (for PostgreSQL or Slony-I; for MySQL it's native) replication system.

Our informal survey of such things suggests that using scripts to write flat files from databases is not as popular as querying a database in real time (and ensuring the database has a proper amount of fault tolerance to handle the fact that a live telecom system is dependent on it).

Static Realtime

Static Realtime was one of the earliest ways that Asterisk configuration could be stored in a database. It is still somewhat useful for storing simple configuration files in a database (which you might normally place in */etc/asterisk*). We don't tend to use it much anymore because Dynamic Realtime is far better for larger sets of data, and the file-based configuration files are more than adequate for smaller configuration settings.

The same rules that apply to flat files on your system still apply when you're using Static Realtime. For example, after making changes to the configuration you still have to run the `module reload` command for the relevant technology (e.g., `*CLI> module reload res_musiconhold.so`).

8 *pgcluster* appears to be a dead project, and Postgres-R appears to be in its infancy, so there may currently be no good solution for master-master replication using PostgreSQL.

9 There are several tutorials on the web describing how to set up replication with MySQL.

When using Static Realtime, we tell Asterisk which files we want to load from the database using the following syntax in the *extconfig.conf* file:

```
; /etc/asterisk/extconfig.conf
[settings]
filename.conf => driver,database[,table]
```

There is no configuration file called *filename.conf*. Instead, use the actual name of the configuration file you are storing in the database. If the table name is not specified, Asterisk will use the name of the file as the table name instead (less the *.conf* part). Also, all settings inside the *extconfig.conf* file should fall under the `[settings]` header. Be aware that you can't load certain files from realtime at all, including *asterisk.conf*, *extconfig.conf*, and *logger.conf*.

The Static Realtime module uses a very specifically formatted table to allow Asterisk to read the various static files from the database. Table 15-1 illustrates the columns as they must be defined in your database.

Table 15-1. Table layout and description of ast_config

Column name	Column type	Description
`id`	Serial, autoincrementing	An autoincrementing unique value for each row in the table.
`cat_metric`	Integer	The weight of the category within the file. A lower metric means it appears higher in the file (see the sidebar on page 281).
`var_metric`	Integer	The weight of an item within a category. A lower metric means it appears higher in the list (see the sidebar on page 281). This is useful for things like codec order in *sip.conf*, or *iax.conf* where you want `disallow=all` to appear first (metric of `0`), followed by `allow=ulaw` (metric of `1`), then `allow=gsm` (metric of `2`).
`filename`	Varchar 128	The filename the module would normally read from the hard drive of your system (e.g., *musiconhold.conf*, *sip.conf*, *iax.conf*).
`category`	Varchar 128	The section name within the file, such as `[general]`. Do not include the square brackets around the name when saving to the database.
`var_name`	Varchar 128	The option on the left side of the equals sign (e.g., `disallow` is the `var_name` in `disallow=all`).
`var_val`	Varchar 128	The value of an option on the right side of the equals sign (e.g., `all` is the `var_val` in `disallow=all`).
`commented`	Integer	Any value other than 0 will evaluate as if it were prefixed with a semicolon in the flat file (commented out).

A Word About Metrics

The metrics in Static Realtime are used to control the order in which objects are read into memory. Think of the `cat_metric` and `var_metric` as the original line numbers in the flat file. A higher `cat_metric` is processed first, because Asterisk matches categories from bottom to top. Within a category, though, a lower `var_metric` is processed first, because Asterisk processes the options top-down (e.g., `disallow=all` should be set to a value lower than the `allow`'s value within a category to make sure it is processed first).

There's not much more to say about Static Realtime. It was very useful in the past, but has now been mostly superseded by Dynamic Realtime. If you want to read more about it, older versions of this book discuss it in more detail.

Dynamic Realtime

The Dynamic Realtime system is used to load objects that may change often, such as PJSIP entities, queues and their members, and voicemail. Likewise, when new records are likely to be added on a regular basis, we can utilize the power of the database to let us load this information on an as-needed basis.

You have already worked extensively with Dynamic Realtime, since that is how we've been working for this entire book, both during installation, and in most of the examples we have worked through.

All of realtime is configured in the */etc/asterisk/extconfig.conf* file; however, Dynamic Realtime has explicitly defined configuration names. All the predefined names should be configured under the `[settings]` header. For example, defining SIP peers is done using the following format:

```
; extconfig.conf
[settings]
sippeers => driver,database[,table]
```

The table name is optional. If it is omitted, Asterisk will use the predefined name (i.e., `sippeers`) to identify the table in which to look up the data.

The sample file *~/src/asterisk-15.<TAB>/configs/samples/extconfig.conf.sample* contains excellent information about Dynamic Realtime.

Storing Call Detail Records

Call detail records (CDR) contain information about calls that have passed through your Asterisk system. They are discussed further in Chapter 21. Storing CDR is a popular use of databases in Asterisk, because it makes them easier to work with. Also,

by placing records into a database you open up many possibilities, including building your own web interface for tracking statistics such as call usage and most-called locations, billing, or phone company invoice verification.

You should always implement CDR storage to a database on any production system (you can always store CDR to a file as well, so there's nothing lost).

Setting the systemname for Globally Unique IDs

A CDR consists of a unique identifier and several fields of information about the call (including the source and destination channel, length of call, last application executed, and so forth). In a clustered set of Asterisk boxes, it is theoretically possible to have duplication among unique identifiers, since each Asterisk system considers only itself. To address this, we can automatically append a system identifier to the front of the unique IDs by adding an option to */etc/asterisk/asterisk.conf*. For each of your boxes, set an identifier by adding something like:

```
[options]
systemname=toronto
```

The best way to store your call detail records is via the `cdr_adaptive_odbc` module. This module allows you to choose which columns of data built into Asterisk are stored in your table, and it permits you to add additional columns that can be populated with the `CDR()` dialplan function. You can even store different parts of CDR data to different tables and databases, if that is required.

To create the table, we have Alembic. The process is almost identical to the one you performed during the system installation, except of course the *.ini* file is different.

```
$ cd ~/src/asterisk-15.<TAB>/contrib/ast-db-manage

$ cp cdr.ini.sample cdr.ini

$ egrep ^sqlalchemy config.ini

sqlalchemy.url = mysql://asterisk:YouNeedAReallyGoodPasswordHereToo@localhost/asterisk
```

The same credentials we used before will also work for CDR.

```
$ sudo vim cdr.ini
```

Add the line you just got back from grep to this file, and save.

```
$ alembic -c ./cdr.ini upgrade head

INFO  [alembic.runtime.setup] Creating new alembic_version_cdr table.
INFO  [alembic.runtime.migration] Running upgrade  -> 210693f3123d, Create CDR table.
INFO  [alembic.runtime.migration] Running upgrade 210693f3123d -> 54cde9847798
```

Alembic doesn't do too much bragging, so the output is terse, but it appears to have completed successfully. Let's check.

```
$ mysql -u asterisk -p

MySQL> describe asterisk.cdr
```

You should get a list of all the fields in the table (which means Alembic was successful). If you get a message like `Table 'asterisk.cdr' doesn't exist`, that indicates Alembic didn't complete the configuration, and you need to review the messages from the Alembic output to see what went wrong (credentials is usually what causes grief here).

Well, that wasn't too hard, eh? The next step is to tell Asterisk to use this new table for CDR going forward.

```
$ sudo -u asterisk touch /etc/asterisk/cdr_adaptive_odbc.conf

$ sudo -u asterisk vim /etc/asterisk/cdr_adaptive_odbc.conf
```

Into this new file, paste the following:

```
[adaptive_connection]
connection=asterisk
table=cdr
```

This is almost too easy, wouldn't you say? Alrighty, now we just have to reload the `ccdr_adaptive_odbc.so` module in Asterisk:

```
$ sudo asterisk -rvvvvvvv

*CLI> module reload cdr_adaptive_odbc.so
```

You can verify that the Adaptive ODBC backend has been loaded by running the following:[10]

```
*CLI> cdr show status

Call Detail Record (CDR) settings
----------------------------------
  Logging:                    Enabled
  Mode:                       Simple
  Log unanswered calls:       No
  Log congestion:             No

* Registered Backends
  -------------------
    cdr-syslog
    Adaptive ODBC
    cdr-custom
    csv
    cdr_manager
```

10 You may see different backends registered, depending on what configuration you have done with other components of the various CDR modules.

Now place a call that gets answered (e.g., using `Playback()`, or `Dial()`ing another channel and answering it). You should get some CDRs stored into your database. You can check by running `SELECT * FROM CDR;` from your database console.

With the basic CDR information stored in the database, you might want to add some additional information to the `cdr` table, such as the route rate. You can use the `ALTER TABLE` directive to add a column called `route_rate` to the table:

```
sql> ALTER TABLE cdr ADD COLUMN route_rate varchar(10);
```

Now reload the `cdr_adaptive_odbc.so` module from the Asterisk console:

```
*CLI> module reload cdr_adaptive_odbc.so
```

and populate the new column from the Asterisk dialplan using the `CDR()` function, like so:

```
exten => _NXXNXXXXXX,1,Verbose(1,Example of adaptive ODBC usage)
   same => n,Set(CDR(route_rate)=0.01)
   same => n,Dial(SIP/my_itsp/${EXTEN})
   same => n,Hangup()
```

After the alteration to your database and dialplan, you can place a call and then look at your CDRs. You should see something like the following:

```
+--------------+----------+---------+------------+
| src          | duration | billsec | route_rate |
+--------------+----------+---------+------------+
| 0000FFFF0008 | 37       | 30      | 0.01       |
+--------------+----------+---------+------------+
```

In reality, storing rating in the call record might not be ideal (CDR is typically used as a raw resource, and things such as rates are added downstream by billing software). The ability to add custom fields to CDR is very useful, but be careful not to use your call records to replace a proper billing platform. Best to keep your CDR clean and do further processing downstream.

Additional Configuration Options for cdr_adaptive_odbc.conf

Some extra configuration options exist in the *cdr_adaptive_odbc.conf* file that may be useful. The first is that you can define multiple databases or tables to store information into, so if you have multiple databases that need the same information, you can simply define them in *res_odbc.conf*, create tables in the databases, and then refer to them in separate sections of the configuration:

```
[mysql_connection]
connection=asterisk_mysql
table=cdr

[mssql_connection]
connection=production_mssql
table=call_records
```

If you specify multiple sections using the same connection and table, you will get duplicate records.

Beyond just configuring multiple connections and tables (which of course may or may not contain the same information; the CDR module we're using is adaptive to situations like that), we can define aliases for the built-in variables, such as `accountcode`, `src`, `dst`, and `billsec`.

If we were to add aliases for column names for our MS SQL connection, we might alter our connection definition like so:

```
[mssql_connection]
connection=production_mssql
table=call_records
alias src => Source
alias dst => Destination
alias accountcode => AccountCode
alias billsec => BillableTime
```

In some situations you may specify a connection where you only want to log calls from a specific source, or to a specific destination. We can do this with filters:

```
[logging_for_device_0000FFFF0008]
connection=asterisk_mysql
table=cdr_for_0000FFFF0008
filter src => 0000FFFF0008
```

If you need to populate a certain column with information based on a section name, you can set it statically with the `static` option, which you may utilize with the `filter` option:

```
[mysql_connection]
connection=asterisk_mysql
table=cdr

[filtered_mysql_connection]
connection=asterisk_mysql
table=cdr
filter src => 0000FFFF0008
static "DoNotCharge" => accountcode
```

In the preceding example, you will get duplicate records in the same table, but all the information will be the same except for the populated `accountcode` column, so you should be able to filter it out using SQL.

Database Integration of ACD Queues

With a Call Center (often referred to as ACD queues), it can be very useful to be able to allow adjustment of queue parameters without having to edit and reload configuration files. Management of a call center can be a complex task, and allowing for simpler adjustment of parameters can make everyone's life a whole lot easier.

The queues themselves we've already placed in the database in Chapter 12. If, however, you also want to store dialplan parameters relating to your queues, the database can do that too.

Storing Dialplan Parameters for a Queue in a Database

The dialplan application `Queue()` allows for several parameters to be passed to it. The CLI command `core show application Queue` defines the following syntax:

```
[Syntax]
Queue(queuename[,options[,URL[,announceoverride[,timeout[,AGI[,macro[,gosub[,
  rule[,position]]]]]]]])
```

Since we're storing our queue in a database, why not also store the parameters you wish to pass to the queue in a similar manner?[11]

```
MySQL> CREATE TABLE `pbx`.`QueueDialplanParameters` (
  `QueueDialplanParametersID` mediumint(8) NOT NULL auto_increment,
  `Description` varchar(128) NOT NULL,
  `QueueID` mediumint(8) unsigned NOT NULL COMMENT 'Pointer to asterisk.queues table',
  `options` varchar(45) default 'n',
  `URL` varchar(256) default NULL,
  `announceoverride` bit(1) default NULL,
  `timeout` varchar(8) default NULL,
  `AGI` varchar(128) default NULL,
  `macro` varchar(128) default NULL,
  `gosub` varchar(128) default NULL,
  `rule` varchar(128) default NULL,
  `position` tinyint(4) default NULL,
  `queue_tableName` varchar(128) NOT NULL,
  PRIMARY KEY  (`QueueDialplanParametersID`)
);
```

Using `func_odbc`, you can write a function that will return the dialplan parameters relevant to that queue:

11 Note that we're creating this table in our `pbx` schema, rather than the `asterisk` schema, and that is because this is not a table that comes with Asterisk, but instead one we're creating ourselves. We recommend letting Asterisk and Alembic have exclusive control over the `asterisk` schema, and using a custom schema (such as `pbx`) for anything custom we might create.

```
[QUEUE_DETAILS]
prefix=GET
dsn=asterisk
readsql=SELECT * FROM pbx.QueueDialplanParameters
readsql+= WHERE QueueDialplanParametersID='${ARG1}'
```

Then pass those parameters to the `Queue()` application as calls arrive:

```
exten => s,1,Verbose(1,Call entering queue named ${SomeValidID)
  same => n,Set(QueueParameters=${GET_QUEUE_DETAILS(SomeValidID)})
  same => n,Queue(${QueueParameters})
```

While somewhat more complicated to develop than just writing an appropriate dialplan, the advantage is that you will be able to manage a larger number of queues, with a wider variety of parameters, using dialplan that is flexible enough to handle any sort of parameters the queueing application in Asterisk accepts. For anything more than a very simple queue, we think you will find the use of a database for all this is well worth the effort.

Writing queue_log to Database

Finally, we can store our `queue_log` to a database, which can make it easier for external applications to extract queue performance details from the system:

```
CREATE TABLE queue_log (
  id int(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  time char(26) default NULL,
  callid varchar(32) NOT NULL default '',
  queuename varchar(32) NOT NULL default '',
  agent varchar(32) NOT NULL default '',
  event varchar(32) NOT NULL default '',
  data1 varchar(100) NOT NULL default '',
  data2 varchar(100) NOT NULL default '',
  data3 varchar(100) NOT NULL default '',
  data4 varchar(100) NOT NULL default '',
  data5 varchar(100) NOT NULL default '',
  PRIMARY KEY (`id`)
);
```

Edit your *extconfig.conf* file to refer to the `queue_log` table:

```
[settings]
queue_log => odbc,asterisk,queue_log
```

A restart of Asterisk, and your queue will now log information to the database. As an example, logging an agent into the sales queue should produce something like this:

```
mysql> select * from queue_log;
+----+----------------------------+---------------------+-----------+
| id | time                       | callid              | queuename |
+----+----------------------------+---------------------+-----------+
|  1 | 2013-01-22 15:07:49.772263 | NONE                | NONE      |
|  2 | 2013-01-22 15:07:49.809028 | toronto-1358885269.1 | support   |
+----+----------------------------+---------------------+-----------+
```

```
+-----------------+------------+-------+-------+-------+-------+-------+
| agent           | event      | data1 | data2 | data3 | data4 | data5 |
+-----------------+------------+-------+-------+-------+-------+-------+
| NONE            | QUEUESTART |       |       |       |       |       |
| SIP/0000FFFF0001 | ADDMEMBER  |       |       |       |       |       |
+-----------------+------------+-------+-------+-------+-------+-------+
```

If you're developing any sort of external application that needs access to queue statistics, having the data stored in this manner will prove far superior to using the */var/log/asterisk/queue_log* file.

Conclusion

In this chapter, you learned about several areas where Asterisk can integrate with a relational database. This is useful for systems where you need to start scaling by clustering multiple Asterisk boxes working with the same centralized information, or when you want to start building external applications to modify information without requiring a reload of the system (i.e., not requiring the modification of flat files).

CHAPTER 16

Introduction to Interactive Voice Response

One day Alice came to a fork in the road and saw a Cheshire cat in a tree. "Which road do I take?" she asked.

"Where do you want to go?" was his response.

"I don't know," Alice answered.

"Then," said the cat, "it doesn't matter."

—Lewis Carroll

The term *Interactive Voice Response* (IVR) is often misused to refer to an automated voice attendant, but the two are very different things. The purpose of an IVR system is to take input from a caller, perform an action based on that input (commonly, looking up data in an external system such as a database), and speak a result to the caller. The purpose of an automated attendant (which we covered in Chapter 14) is to route calls. Originally, an IVR didn't even need to be a telephone system. Anything that took input from a human and spoke back a result fell with the realm of an IVR. Traditionally, IVR systems have been complex, expensive, and annoying to implement. Asterisk changes all that.

Components of an IVR

The most basic elements of an IVR are quite similar to those of an automated attendant, though the goal is different. We need at least one prompt to tell the caller what the IVR expects, a method of receiving input from the caller, logic to verify that the caller's response is valid input, logic to determine what the next step of the IVR should be, and finally, a storage mechanism for the responses, if applicable. We might think of an IVR as a decision tree, although it need not have any branches. For example, a survey may present exactly the same set of prompts to each caller, regardless of

what choices the callers make, and the only routing logic involved is whether the responses given are valid for the questions.

From the caller's perspective, every IVR needs to start with a prompt. This initial prompt will tell the caller what the IVR is for and ask the caller to provide the first input. We discussed prompts in the automated attendant in Chapter 14. Later, we'll create a dialplan that will allow you to better manage multiple voice prompts.

The second component of an IVR is a method for receiving input from the caller. Recall that in Chapter 14 we discussed the `Background()` and `WaitExten()` applications for receiving a new extension. While you could create an IVR using `Background()` and `WaitExten()`, it is generally easier and more practical to use the `Read()` application, which handles both the prompt and the capture of the response. The `Read()` application was designed specifically for use with IVR systems. Its syntax is as follows:

```
Read(variable[,filename[&filename2...]][,maxdigits][,option][,attempts][,timeout])
```

The arguments are described in Table 16-1.

Table 16-1. The Read() application

Argument	Purpose
`variable`	The variable into which the caller's response is stored. It is best practice to give each variable in your IVR a name that is similar to the prompt associated with that variable. This will help later if, for business reasons or ease of use, you need to reorder the steps of the IVR. Naming your variables `var1`, `var2`, etc., may seem easy in the short term, but later in your life cycle it will make fixing bugs more difficult.
`prompt`	A file (or list of files, joined together with the & character) to play for the caller, requesting input. Remember to omit the format extension on the end of each filename.
`maxdigits`	The maximum number of characters to allow as input. In the case of yes/no and multiple-choice questions, it's best practice to limit this value to `1`. In the case of longer lengths, the caller may always terminate input by pressing the # key.

Argument	Purpose

`options`	`s` *(skip)*
	Exit immediately if the channel has not been answered.
	
	`i` *(indication)*
	Rather than playing a prompt, play an indication tone of some sort (such as the dialtone).
	
	`n` *(no answer)*
	Read digits from the caller, even if the line is not yet answered.
	
	`attempts`
	The number of times to play the prompt. If the caller fails to enter anything, the `Read()` application can automatically reprompt the user. The default is one attempt.
	
	`timeout`
	The number of seconds the caller has to enter his input. The default value in Asterisk is 10 seconds, although it can be altered for a single prompt using this option, or for the entire session by assigning a value using the dialplan function `TIMEOUT(response)`.

Once the input is received, it must be validated. If you do not validate the input, you are more likely to find your callers complaining of an unstable application. It is not enough to handle the inputs you are expecting; you also need to handle inputs you do not expect. For example, callers may get frustrated and dial 0 when in your IVR; if you've done a good job, you will handle this gracefully and connect them to somebody who can help them, or provide a useful alternative. A well-designed IVR (just like any program) will try to anticipate every possible input and provide mechanisms to gracefully handle that input.

Once the input is validated, you can submit it to an external resource for processing. This could be done via a database query, a submission to a URI, an AGI program, or many other things. This external application should produce a result, which you will want to relay back to the caller. This could be a detailed result, such as "Your account balance is…" or a simple confirmation, such as "Your account has been updated." We can't think of any real-world case where some sort of result returned to the caller is not required.

Sometimes the IVR may have multiple steps, and therefore a result might include a request for more information from the caller in order to move to the next step of the IVR application.

It is possible to design very complex IVR systems, with dozens or even hundreds of possible paths. We've said it before and we'll say it again: people don't like talking to your phone system, regardless of how clever it is. Keep your IVR simple for your callers, and they are much more likely to get some benefit from it.

A Perfectly Tasty IVR

An excellent example of an IVR that people love to use is one that many pizza delivery outfits use: when you call to place your order, an IVR looks up your caller ID and says "If you would like the exact same order as last time, press 1."

That's all it does, and it's perfect.

Obviously, these companies could design massively complex IVRs that would allow you to select each and every detail of your pie ("for seven-grain crust, press 7"), but how many inebriated, starving customers could successfully navigate something like that at 3 A.M.?

The best IVRs are the ones that require the least input from the caller. Mash that 1 button and your 'za is on its way! Woo hoo!

IVR Design Considerations

When designing your own IVR, there are some important things to keep in mind. We've put together this list of things to do and things not to do in your IVR.

Do

- Keep it simple.
- Have an option to dial 0 to reach a live person.
- Handle errors gracefully.

Don't

- Think that an IVR can completely replace people.
- Use your IVR to show people how clever you are.
- Try to replicate your website with an IVR.
- Bother building an IVR if you can't take numeric or spoken input. Nobody wants to have to spell their name on the dialpad of a phone.[1]
- Force your callers to listen to advertising. Remember that they can hang up at any moment they wish.

1 Especially if it's something like Van Meggelen.

Asterisk Modules for Building IVRs

The "frontend" of the IVR (the parts that interact with the callers) can be handled in the dialplan. It is possible to build an IVR system using the dialplan alone (perhaps with the *astdb* to store and retrieve data); however, you will typically need to communicate with something external to Asterisk (the "backend" of the IVR).

CURL()

The `CURL()` dialplan function in Asterisk allows you to span entire web applications with a single line of dialplan code. We'll use it in our sample IVR later in this chapter.

While you'll find `CURL()` itself to be quite simple to use, the creation of the web application will require experience with web development.

func_odbc

Using `func_odbc`, it is possible to develop extremely complex applications in Asterisk using nothing more than dialplan code and database lookups. If you are not a strong programmer but are very adept with Asterisk dialplans and databases, you'll love `func_odbc` just as much as we do. Check it out in Chapter 15.

AGI

The Asterisk Gateway Interface is such an important part of integrating external applications with Asterisk that we gave it its own chapter. You can find more information in Chapter 18.

AMI

The Asterisk Manager Interface is a socket interface that you can use to get configuration and status information, request actions to be performed, and be notified about things happening to calls. We've written an entire chapter on AMI as well. You can find more information in Chapter 17.

ARI

Asterisk's REST interface builds on knowledge gained over the years about how to integrate Asterisk with current-generation web-centric applications. It is so important, that yes, once again, there is a complete chapter dedicated to it. If you're looking to build complex IVRs using Asterisk, take a closer look at ARI in Chapter 19.

A Simple IVR Using CURL()

Before we go running off writing an external program to handle something, we always give some careful thought about whether there's a way to do the work in the dialplan. One powerful way that Asterisk can interact with external data is through a URL, which the GNU/Linux program cURL does very well. In Asterisk, `CURL()` is a dialplan function.

We're going to use `CURL()` as an example of what an extremely simple IVR can look like. We're going to request our external IP address from *https://ipinfo.io/ip*.[2]

In reality, most IVR applications are going to be far more complex. Even most uses of `CURL()` will tend to be complex, since a URI can return a massive and highly variable amount of data, the vast majority of which will be incomprehensible to Asterisk. The point being that an IVR is not just about the dialplan; it is also very much about the external applications that are triggered by the dialplan, which are doing the real work of the IVR.

The `CURL()` module was installed during our installation process several chapters ago.

The Dialplan

The dialplan for our example IVR is very simple. The `CURL()` function will retrieve our IP address from *https://ipinfo.io/ip*, and then `SayAlpha()` will speak the results to the caller:

```
exten => *764,1,Verbose(2, Run CURL to get IP address from whatismyip.org)
    same => n,Answer()
    same => n,Set(MyIPAddressIs=${CURL(https://ipinfo.io/ip)})
    same => n,SayAlpha(${MyIPAddressIs})
    same => n,Hangup()
```

The simplicity of this is impossibly cool. In a traditional IVR system, this sort of thing could take days to program, assuming it would be possible at all.

A Prompt-Recording IVR Function

In Chapter 14, we created a simple bit of dialplan to record prompts. It was fairly limited in that it only recorded one filename, and thus for each prompt a separate

2 These free IP lookup websites seem to get bought out all the time, and turned into advertising gateways, so what might have worked at this writing may no longer work. What you need is a website that will return your IP address, and nothing else. Today, that seems to be *https://ipinfo.io/ip*. By the time you read this, it may be something else.

extension was needed. Here, we expand upon that to create a complete menu for recording prompts. Since this is a complex bit of dialplan, but it's not a subroutine or a local channel, we're going to create a new section of dialplan for various features, and put things like this there:

```
;FEATURES
[prompts]
exten => s,1,Answer
exten => s,n,Set(step1count=0) ; Initialize counters

; If we get no response after 3 times, we stop asking
   same => n(beginning),GotoIf($[${step1count} > 2]?end)
   same => n,Read(which,prompt-instructions,3)
   same => n,Set(step1count=$[${step1count} + 1])

; All prompts must be 3 digits in length
   same => n,GotoIf($[${LEN(${which})} != 3]?beginning)
   same => n,Set(step1count=0) ; Successful response; reset counters
   same => n,Set(step2count=0)

   same => n(step2),Set(step2count=$[${step2count} + 1])
   same => n,GotoIf($[${step2count} > 2]?beginning) ; No response after 3 tries

; If the file doesn't exist, then don't ask whether to play it
   same => n,GotoIf($[${STAT(f,/var/lib/asterisk/sounds/${which}.wav)} = 0]?recordonly)
   same => n,Background(prompt-tolisten)

   same => n(recordonly),Background(prompt-torecord)
   same => n,WaitExten(10) ; Wait 10 seconds for a response
   same => n,Goto(step2)

   same => n(end),Playback(goodbye)
   same => n,Hangup()

exten => 1,1,Set(step2count=0)
   same => n,Background(/var/lib/asterisk/sounds/${which})
   same => n,Goto(s,step2)

exten => 2,1,Set(step2count=0)
   same => n,Playback(prompt-waitforbeep)
   same => n,Record(${CHANNEL(uniqueid)}.wav)

   same => n(listen),Playback(${CHANNEL(uniqueid)})
   same => n,Set(step3count=0)
   same => n,Read(saveornot,prompt-1tolisten-2tosave-3todiscard,1,,2,3)
   same => n,GotoIf($["${saveornot}" = "1"]?listen)
   same => n,GotoIf($["${saveornot}" = "2"]?saveit)
   same => n,GotoIf($["${saveornot}" = "3"]?tossit)
   same => n,Goto(listen)

   same => n(tossit),System(rm -f /var/lib/asterisk/sounds/${CHANNEL(uniqueid)}.wav)
   same => n,Goto(s,beginning)

   same => n(saveit),Noop('Set' app used to shorten example)
   same => n,Set(PromptToSave=/var/lib/asterisk/sounds/${CHANNEL(uniqueid)}.wav
   same => n,Set(WhereToSave=/var/lib/asterisk/sounds/${which}.wav
   same => n,System(mv -f ${PromptToSave} ${WhereToSave})
```

```
    same => n,Playback(prompt-saved)
    same => n,Goto(s,beginning)
```

In this system, the name of the prompt is no longer descriptive; instead, it is a number. This means that you can record a far greater variety of prompts using the same mechanism, but the trade-off is that your prompts will no longer have descriptive names.

If you want to test this, you'll need to record the prompts this IVR function uses (it's kinda meta, but yup, our prompt creator needs prompts).

Drop this into your dialplan:

```
exten => 510,1,GoSub(subRecordPrompt,${EXTEN},1(prompt-tolisten))     ; press 1
exten => 511,1,GoSub(subRecordPrompt,${EXTEN},1(prompt-torecord))     ; press 2
exten => 512,1,GoSub(subRecordPrompt,${EXTEN},1(prompt-instructions)) ;3-digit (000 to 999)
exten => 513,1,GoSub(subRecordPrompt,${EXTEN},1(prompt-waitforbeep))  ; wait for beep
exten => 514,1,GoSub(subRecordPrompt,${EXTEN},1(prompt-1tolisten-2tosave-3todiscard))
exten => 515,1,GoSub(subRecordPrompt,${EXTEN},1(prompt-saved))
```

Then phone them one by one and record as required.

Once you've recorded the prompts your prompt recorder needs, you should be able to test it out.

```
exten => *742,1,Noop(Prompts)
    same => n,Goto(prompts,s,1)
    same => n,Hangup()
```

From this point forward, you can record prompts using just a numeric identifier. You'll need a way to keep track of what prompt says what, but from a recording perspective you shouldn't need to write more dialplan every time you need a prompt.

Speech Recognition and Text-to-Speech

Although traditionally and still in most cases today, an IVR system presents prerecorded prompts to the caller and accepts input by way of the dialpad, it is also possible to: a) generate prompts artificially, popularly known as *text-to-speech*; and b) accept verbal inputs through a speech recognition engine.

While the concept of being able to have an intelligent conversation with a machine is something sci-fi authors have been promising us for many long years, the actual science of this remains complex and error-prone. Despite their amazing capabilities, computers are ill-suited to the task of appreciating the subtle nuances of human speech.

Having said that, it should be noted that companies such as Google have achieved amazing advances in both text-to-speech and speech recognition. There are now APIs available that can do a remarkable job of making sense out of what is being said to them. Google of course benefits from having a massive backend that can perform

near-miraculous feats of processing; something your IVR might not be able to fully harness.

Text-to-Speech

Text-to-speech (also known as *speech synthesis*) requires that a system be able to artificially construct speech from stored data. While it would be nice if we could simply assign a sound to a letter and have the computer produce each sound as it reads the letters, written language is often not phonetic, and seldom reflects the nuances of speech (English is arguably one of the worst languages in this regard).

There are now excellent APIs available from Google (and others), that will do a very good job of reading back what has been written. As of this writing, it's still very obvious that it's a computer speaking, but it is nevertheless possible to generate system prompts on the fly from text, rather than having to record all prompts in advance. The usefulness of this is difficult to evaluate, since humans are still not interested in talking to your machines; they phoned because they want to talk to you.

Speech Recognition

Since we've managed to convince our computers to talk to us, we naturally want to be able to talk to them as well.[3]

Speech recognition was previously complex and expensive, but Google has recently released an API that allows the enormous power of their speech recognition capabilities to be available to external applications.

Conclusion

Asterisk is an excellent IVR platform. This entire book, in many ways, is teaching you skills that can be applied to IVR development. While the mainstream media only really pays attention to Asterisk as a "free PBX," the reality is that Asterisk is at its most potent when used as an IVR. Within any respectable-sized organization, it is very likely that the Linux system administrators are using Asterisk to solve telecom problems that previously were either unsolvable or impossibly expensive to solve. This is a stealthy revolution, but no less significant for its relative obscurity.

If you are in the IVR business, you need to get to know Asterisk.

3 Actually, most of us talk to our computers, but this is seldom polite.

CHAPTER 17

Asterisk Manager Interface and Call Files

John Malkovich: I have seen a world that NO man should see!
Craig Schwartz: Really? Because for most people it's a rather enjoyable experience.
—Being John Malkovich

The Asterisk Manager Interface (AMI) is a system monitoring and management interface provided by Asterisk. It allows live monitoring of events that occur in the system, as well as enabling requests for Asterisk to perform some action. The available actions are wide-ranging and include things such as returning status information or originating new calls. Many interesting applications have been developed on top of Asterisk that use the AMI as their primary interface to Asterisk.

This chapter also includes documentation on the use of call files. Asterisk's call files are an easy way to originate a few calls. Once call origination volume increases or your needs become otherwise more complex, you can move on to using the AMI. In fact, we find call files to be useful enough that we're going to talk about them first.

Call Files

It is common to use AMI for originating calls, but in many situations it's easier to use call files. A call file is a simple text file that describes the call that you would like Asterisk to originate. When a call file is placed into the */var/spool/asterisk/outgoing* directory, Asterisk will immediately detect that a file has been placed there and process the call.

Asterisk comes with a sample call file, which you will find at *~/src/asterisk-15.<TAB>/sample.call* (or wherever the root directory of your Asterisk source is located).

Your First Call File

For your first call file, let's create a call between two of your telephones. Make sure you have at least two of your phones registered and working. For this example we'll be using `SOFTPHONE_A` and `SOFTPHONE_B`.

Create the following file in your home directory:

```
$ vim ~/call-file

Channel: PJSIP/SOFTPHONE_A
Extension: 103
Context: sets
```

Make a copy of this file (so that you don't have to re-create it every time you want to run it):

```
$ cp ~/call-file docall
```

Change the ownership of the *docall* file to `asterisk`:

```
$ chown asterisk:asterisk docall
```

Move the *docall* file into Asterisk's *outgoing* folder.

```
$ sudo mv docall /var/spool/asterisk/outgoing
```

Sometimes, the easiest way is the best way.

> You'll probably find yourself doing multiple edits on your source call file. You could just move that file you created, rather than making a copy of it first, but then you'd have to re-create it each time you make an edit, and this gets annoying. All that typing can be saved as a one-liner, and run like this:
>
> ```
> $ cp ~/call-file docall \
>  sudo chown asterisk:asterisk docall \
>  sudo mv docall /var/spool/asterisk/outgoing/
> ```
>
> Try it and you'll see how much easier this is than creating and moving a new call file every time.

The use of `mv` instead of `cp` here is important. Asterisk is watching for contents to show up in the *spool* directory. If you use copy, Asterisk may try to read the new file before the contents have been copied into it. Creating a file and then moving it avoids this problem.

Get comfortable with using call files, and you may find them solving problems you'd otherwise have to perform far more work to achieve.

Notes About Call Files

The `Channel` component of the call file is required. Normally, a call coming into Asterisk is initiated by the endpoint (for example, you make a call from your phone). In a call file, that connection has to happen the other way around—Asterisk reaches out to the endpoint, and only when it answers can the call start. Plan accordingly.

You also must specify the `Context` where the call will begin once the initial channel has answered. This can be useful, since it means you can connect the call through a context that wouldn't normally be accessible to that channel, but in practice we'd suggest you simply provide the same context that channel would have entered the dialplan through if it had initiated the call normally.

The `Extension` must of course also be specified. This would typically be the phone number that is to be called, but of course it could be any valid extension within the `Context`.

The rest of the parameters of the call file are optional, and are detailed both in the *~/src/asterisk-15.<TAB>/sample.call* file, and on the Asterisk wiki website.

AMI Quick Start

This section is for getting your hands dirty with the AMI as quickly as possible. First, put the following configuration in */etc/asterisk/manager.conf*:

```
; Turn on the AMI and ask it to only accept connections from localhost.
[general]
enabled = yes
webenabled = yes
bindaddr = 127.0.0.1

; Create an account called "hello", with a password of "world"
[hello]
secret=world
read=all     ; Receive all types of events
write=all    ; Allow this user to execute all actions
```

This sample configuration is set up to allow only local connections to the AMI. If you intend to make this interface available over a network, it is strongly recommended that you only do so using TLS. The use of TLS is discussed in more detail later in this chapter.

Once the AMI configuration is ready, enable the built-in HTTP server by putting the following contents in */etc/asterisk/http.conf*:

```
; Enable the built-in HTTP server, and only listen for connections on localhost.
[general]
enabled = yes
bindaddr = 127.0.0.1
```

Reload the manager and http servers from the Asterisk CLI:

```
*CLI> manager reload

*CLI> module reload http
```

AMI over TCP

There are multiple ways to connect to the AMI, but a TCP socket is the most common. We will use `telnet` to demonstrate AMI connectivity. We'll need to install `telnet` for this:

```
$ sudo yum -y install telnet
```

This example shows these steps:

1. Connect to the AMI over a TCP socket on port 5038.
2. Log in using the `Login` action.
3. Execute the `Ping` action.
4. Log off using the `Logoff` action.

Here's how to do that using `telnet`:

```
$ telnet localhost 5038

Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Asterisk Call Manager/4.0.3
```

You've connected, but it's going to hang up on you unless you authenticate yourself. Paste the following into the `telnet` window:

```
Action: Login
Username: hello
Secret: world
```

Note there needs to be a blank line after the commands (press Enter after you paste everything if nothing happens).

```
Response: Success
Message: Authentication accepted
```

OK, it likes us. Let's run a simple command just to verify it's really talking to us:

```
Action: Ping

Response: Success
Ping: Pong
```

So far, so good. We'll just tidy up and log off now.

```
Action: Logoff

Response: Goodbye
Message: Thanks for all the fish.
Connection closed by foreign host.
```

You have verified that AMI is accepting connections via a TCP connection.

AMI over HTTP

It is also possible to use the AMI over HTTP. We will perform the same actions as before, but over HTTP instead of the native TCP interface to the AMI. AMI over HTTP is covered in more detail in "AMI over HTTP" on page 308.

Accounts used for connecting to the AMI over HTTP are the same accounts configured in */etc/asterisk/manager.conf*.

This example demonstrates how to access the AMI over HTTP, log in, execute the `Ping` action, and log off:

```
$ curl "http://localhost:8088/rawman?action=login&username=hello&secret=world" \
-c /tmp/tempcookie

Response: Success
Message: Authentication accepted

$ curl "http://localhost:8088/rawman?action=ping" -b /tmp/tempcookie

Response: Success
Ping: Pong
Timestamp: 1538871944.474131

$ curl "http://localhost:8088/rawman?action=logoff" -b /tmp/tempcookie

Response: Goodbye
Message: Thanks for all the fish.
```

The HTTP interface to AMI lets you integrate Asterisk call control into a web service.

Configuration

The section "AMI Quick Start" on page 301 showed a very basic set of configuration files to get you started. There are many ways you can fine-tune the configuration of the AMI.

manager.conf

The main configuration file for the AMI is */etc/asterisk/manager.conf*. The `[general]` section contains options that control the overall operation of the AMI. Any other sections in the *manager.conf* file will define accounts for logging in and using the AMI. The sample file contains detailed explanations of the various parameters, and can be found in *~/src/asterisk-15<TAB>/configs/samples/manager.conf.sample*.

If you are going to expose your AMI outside the machine it is running on, you will want to configure TLS connectivity.

The *manager.conf* configuration file also contains the configuration of AMI user accounts. You create an account by adding a section with the username inside square brackets. Within each `[username]` section there are options that can be set that will apply only to that account. The *~/src/asterisk-15<TAB>/configs/samples/manager.conf.sample* file also contains detailed explanations of each of these parameters. Our user named `[hello]`, has the simplest configuration, which is to allow all read and write actions. You should normally create AMI users that are restricted to only the actions necessary to their functioning.

Within the `[username]` section, the `read` and `write` options set which manager actions and manager events a particular user has access to. At this writing there are 20 of them: `all`, `system`, `call`, `log`, `verbose`, `agent`, `user`, `config`, `command`, `dtmf`, `reporting`, `cdr`, `dialplan`, `originate`, `agi`, `cc`, `aoc`, `test`, `security`, and `message`. You will find the *manager.conf.sample* file contains a reference for each of these that is relevant to your release (and, if any are added that have not been listed here, they will be in the sample file).

Take special notice of the `system`, `command`, and `originate` permissions. These permissions grant significant power to any applications that are authorized to use them. Only grant these permissions to applications that you have full control over (and ideally are running on the same box).

http.conf

As we've seen, the Asterisk Manager Interface can be accessed over HTTP as well as TCP. To make that work, a very simple HTTP server is embedded in Asterisk. All of the options relevant to the AMI go in the `[general]` section of */etc/asterisk/http.conf*.

Enabling access to the AMI over HTTP requires both */etc/asterisk/manager.conf* and */etc/asterisk/http.conf*. The AMI must be enabled in *manager.conf* with the `enabled` option set to `yes`, and the *manager.conf* option `webenabled` must be set to `yes` to allow access over HTTP. Finally, the `enabled` option in *http.conf* must be set to `yes` to turn on the HTTP server itself.

The available options will be found in your *~/src/asterisk-15<TAB>/configs/samples/http.conf.sample* file.

Protocol Overview

There are two main types of messages on the Asterisk Manager Interface: manager events and manager actions.

Manager events are one-way messages sent from Asterisk to AMI clients to report something that has occurred on the system (Figure 17-1).

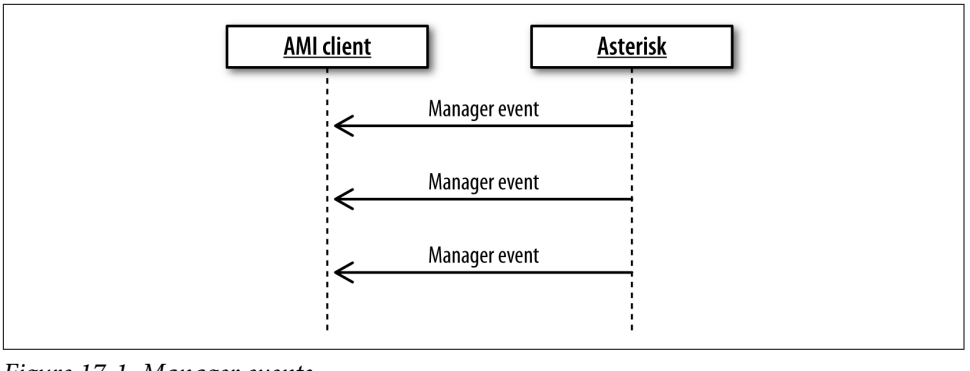

Figure 17-1. Manager events

Manager actions are requests from a client to Asterisk to perform some action and return the result (Figure 17-2). For example, the AMI action `Originate` requests that Asterisk create a new call, and naturally the client application will need responses from Asterisk to indicate the progress of that activity.

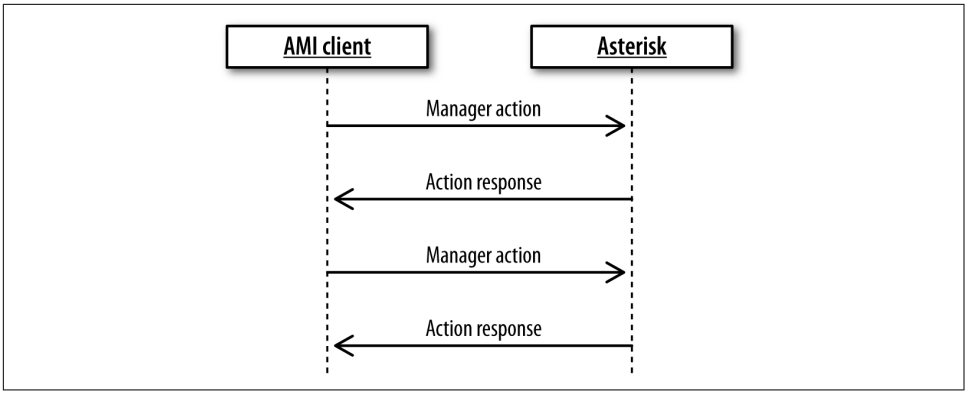

Figure 17-2. Manager actions

Other manager actions are requests for data. For example, there is a manager action to get a list of all active channels on the system: the details about each channel are delivered as a manager event. When the list of results is complete, a final message will be sent to indicate that the end has been reached. See Figure 17-3 for a graphical representation of a client sending this type of manager action and receiving a list of responses.

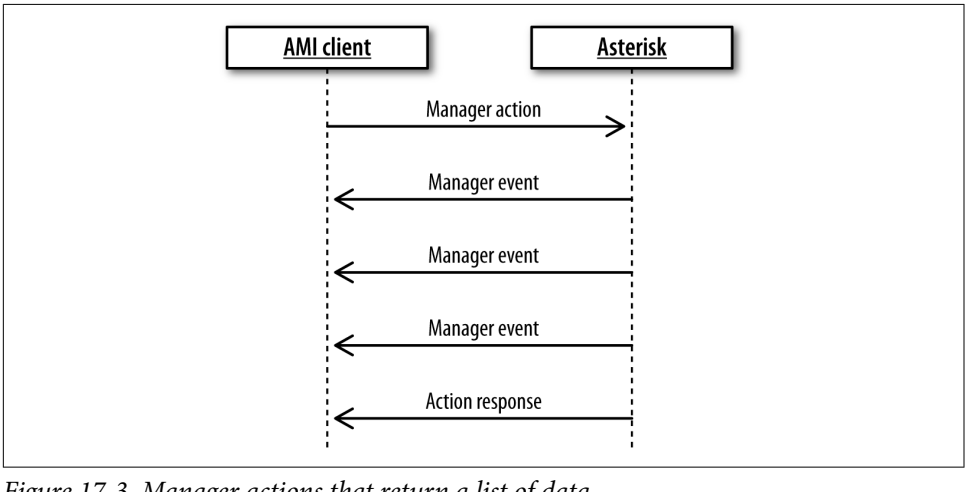

Figure 17-3. Manager actions that return a list of data

Message Encoding

All AMI messages, including manager events, manager actions, and manager action responses, are encoded the same way. The messages are text-based, with lines terminated by a carriage return and a line-feed character. A message is terminated by a blank line:

```
Header1: This is the first header<CR><LF>
Header2: This is the second header<CR><LF>
Header3: This is the last header of this message<CR><LF>
<CR><LF>
```

If you are running tests from a telnet client, what this means is that after the last line of instructions, you'll need to press the Enter key twice.

Events

Manager events always have an `Event` header and a `Privilege` header. The `Event` header gives the name of the event, while the `Privilege` header lists the permission levels associated with the event. Any other headers included with the event are specific to the event type. Here's an example:

```
Event: Hangup
Privilege: call,all
Channel: SIP/0004F2060EB4-00000000
Uniqueid: 1283174108.0
CallerIDNum: 2565551212
CallerIDName: Russell Bryant
Cause: 16
Cause-txt: Normal Clearing
```

The Asterisk CLI includes the commands `manager show events` and `manager show event` *`<event>`*. Run these commands at the Asterisk CLI to get a list of events or to find out the details of a specific event.

Don't forget that an excellent reference for all things Asterisk, including the AMI, is the official Asterisk wiki.

Actions

When executing a manager action, you *must* include the `Action` header. The `Action` header identifies which manager action is being executed. The rest of the headers are arguments to the manager action, and may or may not be required depending on the action.

To get a list of the headers associated with a particular manager action, type **`manager show command`** ***`<Action>`*** at the Asterisk CLI. To get a full list of manager actions supported by the version of Asterisk you are running, enter **`manager show commands`** at the Asterisk CLI.

The final response to a manager action is typically a message that includes the `Response` header. The value of the `Response` header will be `Success` if the manager action was successfully executed. If the manager action was not successfully executed, the value of the `Response` header will be `Error`. For example:

```
Action: Login
Username: hello
Secret: world
```

```
Response: Success
Message: Authentication accepted
```

AMI over HTTP

In addition to the native TCP interface, it is also possible to access the Asterisk Manager Interface over HTTP. Programmers with previous experience writing applications that use web APIs will likely prefer this over the native TCP connectivity. While the TCP interface only offers a single type of message structure, AMI over HTTP offers a few encoding options. You can receive responses in the same format as the TCP interface, in XML, or as a basic HTML page. The encoding type is chosen based on a field in the request URL. The encoding options are discussed in more detail later in this section.

Authentication and session handling

There are two methods of performing authentication against the AMI over HTTP. The first is to use the `Login` action, similar to authentication with the native TCP interface. This is the method that was used in the quick-start example, as seen in "AMI over HTTP" on page 303.

Once successfully authenticated, Asterisk will provide a cookie that identifies the authenticated session. Here is an example response to the `Login` action that includes a session cookie from Asterisk:

```
$ curl -v "http://localhost:8088/rawman?action=login&username=hello&secret=world"
```

The second authentication option is HTTP digest authentication. In this example, the requested encoding type based on the request URL is `rawman`. To indicate that HTTP digest authentication should be used, prefix the encoding type in the request URL with an `a`:

```
$ curl -v --digest -u hello:world http://127.0.0.1:8088/arawman?action=ping
```

/rawman (/arawman) encoding

The `rawman` encoding type is what has been used in all the AMI over HTTP examples in this chapter so far. The responses received from requests using `rawman` are formatted in the exact same way that they would be if the requests were sent over a direct TCP connection to the AMI.

```
curl -v "http://localhost:8088/rawman?action=login&username=hello&secret=world"

curl -v --digest -u hello:world http://127.0.0.1:8088/arawman?action=ping
```

/manager (/amanager) encoding

The `manager` encoding type provides a response in simple HTML form. This interface is primarily useful for experimenting with the AMI:

```
$ curl -v "http://localhost:8088/manager?action=login&username=hello&secret=world"

$ curl -v --digest -u hello:world http://localhost:8088/amanager?action=ping
```

/mxml (/amxml) encoding

The `mxml` encoding type provides responses to manager actions encoded in XML:

```
$ curl -v "http://localhost:8088/mxml?action=login&username=hello&secret=world"

$ curl -v --digest -u hello:world http://localhost:8088/amxml?action=ping
```

Manager events

When connected to the native TCP interface for the AMI, manager events are delivered asynchronously. When using the AMI over HTTP, you must retrieve events by polling for them. You retrieve events over HTTP by executing the `WaitEvent` manager action. The following example shows how events can be retrieved using the `WaitEvent` manager action. The steps are:

1. Start an HTTP AMI session using the `Login` action.
2. Register a SIP phone to Asterisk to generate a manager event.
3. Retrieve the manager event using the `WaitEvent` action.

The interaction looks like this:

```
$ wget --save-cookies cookies.txt \
> "http://localhost:8088/mxml?action=login&username=hello&secret=world" -O -

<ajax-response>
<response type='object' id='unknown'>
    <generic response='Success' message='Authentication accepted' />
</response>
</ajax-response>


$ wget --load-cookies cookies.txt \
< "http://localhost:8088/mxml?action=waitevent" -O -

<ajax-response>
<response type='object' id='unknown'>
    <generic response='Success' message='Waiting for Event completed.' />
</response>
<response type='object' id='unknown'>
    <generic event='PeerStatus' privilege='system,all'
             channeltype='SIP' peer='SIP/0000FFFF0004'
             peerstatus='Registered' address='172.16.0.160:5060' />
</response>
<response type='object' id='unknown'>
```

```
    <generic event='WaitEventComplete' />
</response>
</ajax-response>
```

You'll need to develop mechanisms in your application to ensure that buffered events are frequently polled.

Example Usage

Most of this chapter so far discussed the concepts and configuration related to the AMI. This section will provide some example usage.

Originating a Call

The AMI has the `Originate` manager action that can be used to originate a call. Many of the accepted headers are the same as the options placed in call files. Table 17-1 lists the headers accepted by the `Originate` action.

Table 17-1. Headers for the Originate action

Option	Example value	Description
`ActionID`	`a3a58876-f7c9-4c28-aa97-50d8166f658d`	This header is accepted by most AMI actions. It is used to provide a unique identifier that will also be included in all responses to the action. It gives you a way to identify which request a response is associated with. This is important since all actions, their responses, and events are all transmitted over the same connection (unless using AMI over HTTP).
`Channel`	`SIP/myphone`	This header is critical and must be specified. This describes the outbound call that will be originated. The value is the same syntax that would be used for the channel argument to the `Dial()` application in the dialplan.
`Context`	`default`	This header is used to specify a location in the dialplan to start executing once the outbound call has answered. The `Context`, `Exten`, and `Priority` headers must be used together. When using these headers, the `Application` and `Data` headers should not be used.
`Exten`	`s`	See the documentation for the `Context` header.
`Priority`	`1`	See the documentation for the `Context` header.
`Application`	`ConfBridge`	The `Application` and `Data` headers can be used instead of the `Context`, `Exten`, and `Priority` headers. In this case, the outbound call is directly connected to a single application once the call has been answered.
`Data`	`500`	See the documentation for the `Application` header.
`Timeout`	`30000`	This header specifies how long to wait in milliseconds for an answer before giving up on an outbound call. The default is `30000` milliseconds (30 seconds).
`CallerID`	`Matthew Jordan <(555) 867-5309>`	This header can be used to specify the caller ID used for the outbound call.
`Account`	`someaccount`	This header sets the CDR account code for the outbound call.

Option	Example value	Description
`Variable`	`VARIABLE=VALUE` or `FUNCTION(arguments)=VALUE`	The `Variable` header can be used to set both channel variables or channel functions on the outbound channel. It can be specified multiple times.
`Codecs`	`ulaw,alaw`	This option can be used to limit which codecs are allowed for the outbound call. If not specified, the set of codecs configured in the channel driver configuration file will still be honored.
`EarlyMedia`	`true`	If this header is specified and set to `true`, the outbound call will get connected to the specified extension or application as soon as there is any early media.
`Async`	`true`	If this header is specified and set to `true`, this call will be originated asynchronously. This will allow you to continue executing other actions on the AMI connection while the call is being processed.

The simplest example of using the `Originate` action is via `telnet`:

```
$ telnet localhost 5038

Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Asterisk Call Manager/4.0.3
```

Once the connection is established, you need to log in.

```
Action: Login
Username: hello
Secret: world

Response: Success
Message: Authentication accepted
```

Now you're ready to originate your call. We're doing essentially the same thing we did with the call file, only this time using the AMI:

```
Action: Originate
Channel: PJSIP/SOFTPHONE_A
Context: sets
Exten: 103
Priority: 1
```

You should hear `SOFTPHONE_A` ringing. As soon as you answer it, a call will be placed to `SOFTPHONE_B`.

AMI is no longer involved in what's going on. You can disconnect and the call will continue (leave the call up for now, as we're going to work with the in-progress call next).

```
Action: Logoff

Response: Goodbye
Message: Thanks for all the fish.
Connection closed by foreign host.
```

If you've already hung up the call, that's no problem. You'll just need to re-establish the call, which of course you can do simply by calling one extension from the other (101 to 103, or whatever you want).

Redirecting a Call

Redirecting (or transferring) a call from the AMI is another feature worth mentioning. The `Redirect` AMI action can be used to send one or two channels to any other extension in the Asterisk dialplan. If you need to redirect two channels that are bridged together, do them both at the same time. Otherwise, once one channel has been redirected, the other will be hung up.

An important thing to understand about Asterisk channels is that they don't exist until a call is in progress. The name we all think of as the channel name (e.g., `SOFTPHONE_A`) is not in fact the channel name, but merely a reference to data that is used to create a channel. The naming of a channel takes place when a call is originated (which is when the channel is actually created). What all this means is that you have to determine the full name of the channel before you can act on it.

Originate a call, and then review the `Event: Newchannel` and you will see the channel name under the `Channel:` header.

```
Action: Originate
Channel: PJSIP/SOFTPHONE_A
Context: sets
Exten: 103
Priority: 1

Response: Success
Message: Originate successfully queued

Event: Newchannel
Privilege: call,all
Channel: PJSIP/SOFTPHONE_A-00000013
ChannelState: 0
ChannelStateDesc: Down
CallerIDNum: <unknown>
CallerIDName: <unknown>
ConnectedLineNum: <unknown>
ConnectedLineName: <unknown>
Language: en
AccountCode:
Context: sets
Exten: s
Priority: 1
Uniqueid: 1538939479.29
Linkedid: 1538939479.29
```

The `Newchannel` event will provide the name of the created channel, which in this example is `PJSIP/SOFTPHONE_A-00000013`.

You will need to keep track of these channel names if you wish to properly perform actions on calls in progress. Once the call ends, the channel is destroyed. A new call using the same endpoint will be assigned a different channel name. One channel definition can support multiple calls (for example, multiple calls to a phone are possible), and this is why the channel name is different from the channel definition.

You can redirect a single channel (the other channel will be disconnected):

```
Action: Redirect
Channel: PJSIP/SOFTPHONE_A-00000013
Exten: 209
Context: sets
Priority: 1
```

Or you can redirect two channels:

```
Action: Redirect
Channel: PJSIP/SOFTPHONE_A-00000015
Context: sets
Exten: 209
Priority: 1
ExtraChannel: PJSIP/SOFTPHONE_B-00000016
ExtraContext: sets
ExtraExten: 209
ExtraPriority: 1
```

The redirect function allows you to create powerful external applications that can control calls in progress.

Development Frameworks

Many application developers write code that directly interfaces with the AMI. However, there are a number of frameworks that have been created with the purpose of making AMI application development easier. If you search for Asterisk frameworks in the popular programming language of your choice, you are likely to find one. The onus is on you to determine the suitability of the framework you are interested in. Some things you should look for in a framework include:

Maturity
: Has this project been around for a few years? A mature project is far less likely to have serious bugs in it.

Maintenance
: Check the age of the latest update. If the project hasn't been updated in five years, there's a strong possibility it has been abandoned. It might still be usable, but you'll be on your own. Similarly, what does the bug tracker look like? Are there a lot of important bugs being ignored? (Be discerning here, since often the realities of maintaining a free project require disciplined triage—not everybody's features are going to get added.)

Quality of the code
: Is this a well-written framework? If it was not engineered well, you should be aware of that when deciding whether to trust your project to it.

Community
: Is there an active community of developers using this project? It's likely you'll need help; will it be available when you need it?

Documentation
: The code should be well commented, but ideally, a wiki or other official documentation to support the library is essential.

Table 17-2 lists some frameworks we have found that, as of this writing, met the preceding criteria. There may be others out there.

Table 17-2. AMI development frameworks

Framework	Language
Adhearsion	Ruby
StarPy	Python
Asterisk-Java	Java
AsterNET	.NET
ami-io	Node.js
panoramisk	Python

Conclusion

The Asterisk Manager Interface provides an API for monitoring events from an Asterisk system, as well as requesting that Asterisk perform a wide range of actions. An HTTP interface has been provided, and a number of frameworks have been developed, that make it easier to develop applications.

CHAPTER 18
Asterisk Gateway Interface

Caffeine. The gateway drug.
—Eddie Vedder

The Asterisk dialplan has evolved into a simple yet powerful programming interface for call handling. Many people, however, especially those with a programming background, prefer to implement call handling in a traditional programming language. The Asterisk Gateway Interface (AGI) allows for the development of first-party call control in the programming language of your choice.

Quick Start

This section gives a quick example of using the AGI.

First, let's create the script that we're going to run. AGI scripts are usually placed in */var/lib/asterisk/agi-bin*.

```
$ cd /var/lib/asterisk/agi-bin

$ vim hello-world.sh

#!/bin/bash

# Consume all variables sent by Asterisk
while read VAR && [ -n ${VAR} ] ; do : ; done

# Answer the call.
echo "ANSWER"
read RESPONSE

# Say the letters of "Hello World"
echo 'SAY ALPHA "Hello World" ""'
read RESPONSE

exit 0
```

```
$ chown asterisk:asterisk hello-world.sh

$ chmod 700 hello-world.sh
```

Now, add the following line to */etc/asterisk/extensions.conf*, in your `[sets]` context:

```
exten => 237,1,AGI(hello-world.sh)
```

Save and reload your dialplan, and when you call extension 237 you should hear Allison spell out "Hello World."

AGI Variants

There are a few variants of AGI that differ primarily in the method used to communicate with Asterisk. It is good to be aware of all the options so you can make the best choice based on the needs of your application.

Process-Based AGI

Process-based AGI is the simplest variant of AGI. The quick-start example at the beginning of this chapter is an example of a process-based AGI script. The script is invoked using the `AGI()` application from the Asterisk dialplan. The application to run is specified as the first argument to `AGI()`. Unless a full path is specified, the application is expected to be in the */var/lib/asterisk/agi-bin* directory. Arguments to be passed to your AGI application can be specified as additional arguments to the `AGI()` application in the Asterisk dialplan. The syntax is:

```
AGI(command[,arg1[,arg2[,...]]])
```

Ensure that your application has the proper permissions set so that the Asterisk process user has permissions to execute it. Otherwise, `AGI()` will fail.

Once Asterisk executes your AGI application, communication between Asterisk and your application will take place over `stdin` and `stdout`. More details about this communication will be covered in "AGI Communication Overview" on page 319. For more details about invoking `AGI()` from the dialplan, check the documentation built into Asterisk:

```
*CLI> core show application AGI
```

Pros of process-based AGI

It is the simplest form of AGI to implement.

Cons of process-based AGI
: It is the least efficient form of AGI with regard to resource consumption. Systems with high load should consider FastAGI, discussed in "FastAGI—AGI over TCP" on page 317, instead.

EAGI

EAGI (Enhanced AGI) is a slight variant on `AGI()`. It is invoked in the Asterisk dialplan as `EAGI()`. The difference is that in addition to the communication on `stdin` and `stdout`, Asterisk also provides a unidirectional stream of audio coming from the channel on file descriptor 3. For more details on how to invoke `EAGI()` from the Asterisk dialplan, check the documentation built into Asterisk:

```
*CLI> core show application EAGI
```

Pros of Enhanced AGI
: It has the simplicity of process-based AGI, with the addition of a simple read-only stream of the channel's audio. This is the only variant that offers this feature.

Cons of Enhanced AGI
: Since a new process must be spawned to run your application for every call, it has the same efficiency concerns as regular, process-based AGI.

For an alternative way of gaining access to the audio outside Asterisk, consider using JACK. Asterisk has a module for JACK integration, called `app_jack`. It provides the `JACK()` dialplan application and the `JACK_HOOK()` dialplan function.

FastAGI—AGI over TCP

FastAGI is the term used for AGI call control over a TCP connection. With process-based AGI, an instance of an AGI application is executed on the system for every call, and communication with that application is done over `stdin` and `stdout`. With FastAGI, a TCP connection is made to a FastAGI server. Call control is done using the same AGI protocol, but the communication is over the TCP connection and does not require a new process to be started for every call. The AGI protocol is discussed in more detail in "AGI Communication Overview" on page 319. Using FastAGI is much more scalable than process-based AGI, though it is also more complex to implement.

To use FastAGI you invoke the `AGI()` application in the Asterisk dialplan, but instead of providing the name of the application to execute, you provide an `agi://` URL. For example:

```
exten => 238,1,AGI(agi://127.0.0.1)
```

The default port number for a FastAGI connection is `4573`. A different port number can be appended to the URL after a colon. For example:

```
exten => 238,1,AGI(agi://127.0.0.1:4574)
```

Just as with process-based AGI, arguments can be passed to a FastAGI application. To do so, add them as additional arguments to the `AGI()` application, delimited by commas:

```
exten => 238,1,AGI(agi://192.168.1.199,arg1,arg2,arg3)
```

FastAGI also supports the usage of DNS SRV records, if you provide a URL in the form of `hagi://`. By using SRV records, your DNS servers can return multiple hosts that Asterisk can attempt to connect to. This can be used for high availability and load balancing. In the following example, to find a FastAGI server to connect to, Asterisk will perform a DNS lookup for `_agi._tcp.shifteight.org`:

```
exten => 238,1,AGI(hagi://shifteight.org)
```

In this example, the DNS servers for the `shifteight.org` domain would need at least one SRV record configured for `_agi._tcp.shifteight.org`.

Pros of FastAGI

It's more efficient than process-based AGI. Rather than spawning a process per call, a FastAGI server can be built to handle many calls.

DNS can be used to achieve high availability and load balancing among FastAGI servers to further enhance scalability.

Cons of FastAGI

It is more complex to implement a FastAGI server than to implement a process-based AGI application.

Async AGI—AMI-Controlled AGI

Async AGI allows an application that uses the Asterisk Manager Interface (AMI) to asynchronously queue up AGI commands to be executed on a channel. This can be especially useful if you are already making extensive use of the AMI and would like to enhance your application to handle call control, rather than writing a detailed Asterisk dialplan or developing a separate FastAGI server.

More information on the Asterisk Manager Interface can be found in Chapter 17.

Async AGI is invoked by the `AGI()` application in the Asterisk dialplan. The argument to `AGI()` should be `agi:async`, as shown in the following example:

```
exten => 239,AGI(agi:async)
```

Additional information on how to use async AGI over the AMI can be found in the next section.

Pros of async AGI
: An existing AMI application can be used to control calls using AGI commands.

Cons of async AGI
: It is the most complex way to implement AGI.

Setting Up /etc/asterisk/manager.conf for Async AGI

To make use of async AGI, an AMI account must have the `agi` permission for both `read` and `write`. For example, the following user defined in *manager.conf* would be able to both a) execute AGI manager actions, and b) receive AGI manager events:

```
; Define a user called 'hello', with a password of 'world'.
; Give this user read/write permissions for AGI.
;
[hello]
secret = world
read = agi
write = agi
```

AGI Communication Overview

The preceding section discussed the variations of AGI that can be used. This section goes into more detail about how your custom AGI application communicates with Asterisk once `AGI()` has been invoked.

Setting Up an AGI Session

Once `AGI()` or `EAGI()` has been invoked from the Asterisk dialplan, some information is passed to the AGI application to set up the AGI session. This section discusses what steps are taken at the beginning of an AGI session for the different variants of AGI.

Process-based AGI/FastAGI

For a process-based AGI application or a connection to a FastAGI server, the variables listed in Table 18-1 will be the first pieces of information sent from Asterisk to your application. Each variable will be on its own line, in the form:

```
agi_variable: value
```

Table 18-1. AGI environment variables

Variable	Value/example	Description
agi_request	hello-world.sh	The first argument that was passed to the AGI() or EAGI() application. For process-based AGI, this is the name of the AGI application that has been executed. For FastAGI, this would be the URL that was used to reach the FastAGI server.
agi_channel	SIP/0004F2060EB4-00000009	The name of the channel that has executed the AGI() or EAGI() application.
agi_language	en	The language set on agi_channel.
agi_type	SIP	The channel type for agi_channel.
agi_uniqueid	1284382003.9	The uniqueid of agi_channel.
agi_version	1.8.0-beta4	The Asterisk version in use.
agi_callerid	12565551212	The full caller ID string that is set on agi_channel.
agi_calleridname	Russell Bryant	The caller ID name that is set on agi_channel.
agi_callingpres	0	The caller presentation associated with the caller ID set on agi_channel. For more information, see the output of core show function CALLERPRES at the Asterisk CLI.
agi_callingani2	0	The caller ANI2 associated with agi_channel.
agi_callington	0	The caller ID TON (Type of Number) associated with agi_channel.
agi_callingtns	0	The dialed number TNS (Transit Network Select) associated with agi_channel.
agi_dnid	7010	The dialed number associated with agi_channel.
agi_rdnis	unknown	The redirecting number associated with agi_channel.
agi_context	phones	The context of the dialplan that agi_channel was in when it executed the AGI() or EAGI() application.
agi_extension	500	The extension in the dialplan that agi_channel was executing when it ran the AGI() or EAGI() application.
agi_priority	1	The priority of agi_extension in agi_context that executed AGI() or EAGI().
agi_enhanced	0.0	An indication of whether AGI() or EAGI() was used from the dialplan. 0.0 indicates that AGI() was used. 1.0 indicates that EAGI() was used.
agi_accountcode	myaccount	The accountcode associated with agi_channel.
agi_threadid	140071216785168	The threadid of the thread in Asterisk that is running the AGI() or EAGI() application. This may be useful for associating logs generated by the AGI application with logs generated by Asterisk, since the Asterisk logs contain thread IDs.
agi_arg_*<argument number>*	my argument	These variables provide the contents of the additional arguments provided to the AGI() or EAGI() application.

For an example of the variables that might be sent to an AGI application, see the AGI communication debug output in "Quick Start" on page 315. The end of the list of variables will be indicated by a blank line. The code handles these variables by reading lines of input in a loop until a blank line is received. At that point, the application continues and begins executing AGI commands.

Async AGI

When you use async AGI, Asterisk will send out a manager event called `AsyncAGI` to initiate the async AGI session. This event will allow applications listening to manager events to take over control of the call via the AGI manager action. Here is an example manager event sent out by Asterisk:

```
Event: AsyncAGI
Privilege: agi,all
SubEvent: Start
Channel: SIP/0000FFFF0001-00000000
Env: agi_request%3A%20async%0Aagi_channel%3A%20SIP%2F0000FFFF0001-00000000%0A \
     agi_language%3A%20en%0Aagi_type%3A%20SIP%0A \
     agi_uniqueid%3A%201285219743.0%0A \
     agi_version%3A%201.8.0-beta5%0Aagi_callerid%3A%2012565551111%0A \
     agi_calleridname%3A%20Julie%20Bryant%0Aagi_callingpres%3A%200%0A \
     agi_callingani2%3A%200%0Aagi_callington%3A%200%0Aagi_callingtns%3A%200%0A \
     agi_dnid%3A%20111%0Aagi_rdnis%3A%20unknown%0Aagi_context%3A%20LocalSets%0A \
     agi_extension%3A%20111%0Aagi_priority%3A%201%0Aagi_enhanced%3A%200.0%0A \
     agi_accountcode%3A%20%0Aagi_threadid%3A%20-1339524208%0A%0A
```

The value of the `Env` header in this `AsyncAGI` manager event is all on one line. The long value of the `Env` header has been URI encoded.

Commands and Responses

Once an AGI session has been set up, Asterisk begins performing call processing in response to commands sent from the AGI application. As soon as an AGI command has been issued to Asterisk, no further commands will be processed on that channel until the current command has been completed. When it finishes processing a command, Asterisk will respond with the result.

The AGI processes commands in a serial manner. Once a command has been executed, no further commands can be executed until Asterisk has returned a response. Some commands can take a very long time to execute. For example, the `EXEC` AGI command executes an Asterisk application. If the command is `EXEC Dial`, AGI communication is blocked until the call is done. If your AGI application needs to interact further with Asterisk at this point, it can do so using the AMI, which is covered in Chapter 17.

You can retrieve a full list of available AGI commands from the Asterisk console by running the command `agi show commands`. These commands are described in Table 18-2. To get more detailed information on a specific AGI command, including syntax information for any arguments that a command expects, use `agi show commands topic` *`COMMAND`*. For example, to see the built-in documentation for the `ANSWER` AGI command, you would use `agi show commands topic ANSWER`.

Table 18-2. AGI commands

AGI command	Description
`ANSWER`	Answer the incoming call.
`ASYNCAGI BREAK`	End an async AGI session and have the channel return to the Asterisk dialplan.
`CHANNEL STATUS`	Retrieve the status of the channel. This is used to retrieve the current state of the channel, such as up (answered), down (hung up), or ringing.
`DATABASE DEL`	Delete a key/value pair from the built-in AstDB.
`DATABASE DELTREE`	Delete a tree of key/value pairs from the built-in AstDB.
`DATABASE GET`	Retrieve the value for a key in the AstDB.
`DATABASE PUT`	Set the value for a key in the AstDB.
`EXEC`	Execute an Asterisk dialplan application on the channel. This command is very powerful in that between `EXEC` and `GET FULL VARIABLE`, you can do anything with the call that you can do from the Asterisk dialplan.
`GET DATA`	Read digits from the caller.
`GET FULL VARIABLE`	Evaluate an Asterisk dialplan expression. You can send a string that contains variables and/or dialplan functions, and Asterisk will return the result after making the appropriate substitutions. This command is very powerful in that between `EXEC` and `GET FULL VARIABLE`, you can do anything with the call that you can do from the Asterisk dialplan.
`GET OPTION`	Stream a sound file while waiting for a digit from the caller. This is similar to the `Background()` dialplan application.
`GET VARIABLE`	Retrieve the value of a channel variable.
`HANGUP`	Hang up the channel.[a]
`NOOP`	Do nothing. You will get a result response from this command, just like any other. It can be used as a simple test of the communication path with Asterisk.
`RECEIVE CHAR`	Receive a single character. This only works for channel types that support it, such as IAX2 using `TEXT` frames or SIP using the `MESSAGE` method.
`RECEIVE TEXT`	Receive a text message. This only works in the same cases as `RECEIVE CHAR`.

AGI command	Description
RECORD FILE	Record the audio from the caller to a file. This is a blocking operation similar to the Record() dialplan application. To record a call in the background while you perform other operations, use EXEC Monitor or EXEC MixMonitor.
SAY ALPHA	Say a string of characters. You can find an example of this in "Quick Start" on page 315. To get localized handling of this and the other SAY commands, set the channel language either in the device configuration file (e.g., *sip.conf*) or in the dialplan, by setting the CHANNEL(language) dialplan function.
SAY DIGITS	Say a string of digits. For example, 100 would be said as "one zero zero" if the channel's language is set to English.
SAY NUMBER	Say a number. For example, 100 would be said as "one hundred" if the channel's language is set to English.
SAY PHONETIC	Say a string of characters, but use a common word for each letter (Alpha, Bravo, Charlie…).
SAY DATE	Say a given date.
SAY TIME	Say a given time.
SAY DATETIME	Say a given date and time using a specified format.
SEND IMAGE	Send an image to a channel. IAX2 supports this, but there are no actively developed IAX2 clients that support it that we know of.
SEND TEXT	Send text to a channel that supports it. This can be used with SIP and IAX2 channels, at least.
SET AUTOHANGUP	Schedule the channel to be hung up at a specified point in time in the future.
SET CALLERID	Set the caller ID name and number on the channel.
SET CONTEXT	Set the current dialplan context on the channel.
SET EXTENSION	Set the current dialplan extension on the channel.
SET MUSIC	Start or stop music on hold on the channel.
SET PRIORITY	Set the current dialplan priority on the channel.
SET VARIABLE	Set a channel variable to a given value.
STREAM FILE	Stream the contents of a file to a channel.
CONTROL STREAM FILE	Stream the contents of a file to a channel, but also allow the channel to control the stream. For example, the channel can pause, rewind, or fast-forward the stream.
TDD MODE	Toggle the TDD (Telecommunications Device for the Deaf) mode on the channel.
VERBOSE	Send a message to the verbose logger channel. Verbose messages show up on the Asterisk console if the verbose setting is high enough. Verbose messages will also go to any logfile that has been configured for the verbose logger channel in */etc/asterisk/logger.conf*.
WAIT FOR DIGIT	Wait for the caller to press a digit.
SPEECH CREATE	Initialize speech recognition. This must be done before using other speech AGI commands.[b]
SPEECH SET	Set a speech engine setting. The settings that are available are specific to the speech recognition engine in use.
SPEECH DESTROY	Destroy resources that were allocated for doing speech recognition. This command should be the last speech command executed.
SPEECH LOAD GRAMMAR	Load a grammar.
SPEECH UNLOAD GRAMMAR	Unload a grammar.

AGI command	Description
SPEECH ACTIVATE GRAMMAR	Activate a grammar that has been loaded.
SPEECH DEACTIVATE GRAMMAR	Deactivate a grammar.
SPEECH RECOGNIZE	Play a prompt and perform speech recognition, as well as wait for digits to be pressed.
GOSUB	Execute a dialplan subroutine. This will perform in the same way as the `GoSub()` dialplan application.

[a] When the HANGUP AGI command is used, the channel is not immediately hung up. Instead, the channel is marked as needing to be hung up. Your AGI application must exit first before Asterisk will continue and perform the actual hangup process.

[b] While Asterisk includes a core API for handling speech recognition, it does not come with a module that provides a speech recognition engine. Digium currently provides two commercial options for speech recognition: Lumenvox and Vestec.

Process-based AGI/FastAGI

AGI commands are sent to Asterisk on a single line. The line must end with a single newline character. Once a command has been sent to Asterisk, no further commands will be processed until the last command has finished and a response has been sent back to the AGI application. Here is an example response to an AGI command:

```
200 result=0
```

The Asterisk console allows debugging the communications with an AGI application. To enable AGI communication debugging, run the `agi set debug on` command. To turn debugging off, use `agi set debug off`. While this debugging mode is on, all communication to and from an AGI application will be printed out to the Asterisk console. An example of this output can be found in "Quick Start" on page 315.

Async AGI

When you're using async AGI, you issue commands by using the AGI manager action. To see the built-in documentation for the AGI manager action, run `manager show command AGI` at the Asterisk CLI. A demonstration will help clarify how AGI commands are executed using the async AGI method. First, an extension is created in the dialplan that runs an async AGI session on a channel:

```
exten => 240,AGI(agi:async)
```

When the AGI dialplan application is executed, a manager event called `AsyncAGI` will be sent out with all the AGI environment variables. Details about this event are in "Async AGI" on page 321. After this, AGI manager actions can start to take place via AMI.

The following shows an example manager-action execution and the manager events that are emitted during async AGI processing. After the initial execution of the `AGI` manager action, there is an immediate response to indicate that the command has been queued up for execution. Later, there is a manager event that indicates that the queued command has been executed. The `CommandID` header can be used to associate the initial request with the event that indicates that the command has been executed:

```
Action: AGI
Channel: SIP/0004F2060EB4-00000013
ActionID: my-action-id
CommandID: my-command-id
Command:  VERBOSE "Puppies like cotton candy." 1

Response: Success
ActionID: my-action-id
Message: Added AGI command to queue

Event: AsyncAGI
Privilege: agi,all
SubEvent: Exec
Channel: SIP/0004F2060EB4-00000013
CommandID: my-command-id
Result: 200%20result%3D1%0A
```

The following output is what was seen on the Asterisk console during this async AGI session:

```
    -- Executing [7011@phones:1] AGI("SIP/0004F2060EB4-00000013",
       "agi:async") in new stack
 agi:async: Puppies like cotton candy.
  == Spawn extension (phones, 7011, 1)
exited non-zero on 'SIP/0004F2060EB4-00000013'
```

Ending an AGI Session

An AGI session ends when your AGI application is ready for it to end. The details about how this happens depend on whether your application is using process-based AGI, FastAGI, or async AGI.

Process-based AGI/FastAGI

Your AGI application may exit or close its connection at any time. As long as the channel has not hung up before your application ends, dialplan execution will continue.

If channel hangup occurs while your AGI session is still active, Asterisk will provide notification that this has occurred so that your application can adjust its operation as appropriate.

If a channel hangs up while your AGI application is still executing, a couple of things will happen. If an AGI command is in the middle of executing, you may receive a result code of `-1`. You should not depend on this, though, since not all AGI

commands require channel interaction. If the command being executed does not require channel interaction, the result will not reflect the hangup.

The next thing that happens after a channel hangs up is that an explicit notification of the hangup is sent to your application. For process-based AGI, the signal `SIGHUP` will be sent to the process to notify it of the hangup. For a FastAGI connection, Asterisk will send a line containing the word `HANGUP`.

> If you would like to disable having Asterisk send the `SIGHUP` signal to your process-based AGI application or the `HANGUP` string to your FastAGI server, you can do so by setting the `AGISIGHUP` channel variable, as demonstrated in this short example:
>
> ```
> ; no SIGHUP (AGI) or HANGUP (FastAGI)
> exten => 237,1,Set(AGISIGHUP=no)
>     same => n,AGI(hello-world.sh)
> ```

Once the hangup has happened, the only AGI commands that may be used are those that do not require channel interaction. The documentation for the AGI commands built into Asterisk includes an indication of whether or not each command can be used once the channel has been hung up.

Async AGI

When you're using async AGI, the manager interface provides mechanisms to notify you about channel hangups. When you would like to end an async AGI session for a channel, you must execute the `ASYNCAGI BREAK` command. When the async AGI session ends, Asterisk will send an `AsyncAGI` manager event with a `SubEvent` of `End`. The following is an example of ending an async AGI session:

```
Action: AGI
Channel: SIP/0004F2060EB4-0000001b
ActionID: my-action-id
CommandID: my-command-id
Command: ASYNCAGI BREAK

Response: Success
ActionID: my-action-id
Message: Added AGI command to queue

Event: AsyncAGI
Privilege: agi,all
SubEvent: End
Channel: SIP/0004F2060EB4-0000001b
```

At this point, the channel returns to the next step in the Asterisk dialplan (assuming it has not yet been hung up).

Example: Account Database Access

Example 18-1 is an example of an AGI script. To run this script you would first place it in the */var/lib/asterisk/agi-bin* directory. Then you would execute it from the Asterisk dialplan like this:

```
exten => 241,1,AGI(account-lookup.py)
  same => n,Hangup()
```

This example is written in Python and is very sparsely documented for brevity. It demonstrates how an AGI script interfaces with Asterisk using `stdin` and `stdout`.

The script prompts a user to enter an account number, and then plays back a value associated with that number. In the interest of brevity, we have hardcoded a few fake accounts into the script—this would obviously be something normally handled by a database connection.

The script is intentionally terse, since we are interested in briefly showing some AGI functions without filling this book with pages of code.

Example 18-1. account-lookup.py

```
#!/usr/bin/env python
# An example for AGI (Asterisk Gateway Interface).

import sys

def agi_command(cmd):
    '''Write out the command and return the response'''
    print cmd
    sys.stdout.flush() #clear the buffer
    return sys.stdin.readline().strip() # strip whitespace

asterisk_env = {} # read AGI env vars from Asterisk
while True:
    line = sys.stdin.readline().strip()
    if not len(line):
        break
    var_name, var_value = line.split(':', 1)
    asterisk_env[var_name] = var_value

# Fake "database" of accounts.
ACCOUNTS = {
    '12345678': {'balance': '50'},
    '11223344': {'balance': '10'},
    '87654321': {'balance': '100'},
}

response = agi_command('ANSWER')

# three arguments: prompt, timeout, maxlength
response = agi_command('GET DATA enter_account 3000 8')

if 'timeout' in response:
```

```
    response = agi_command('STREAM FILE goodbye ""')
    sys.exit(0)

# The response will look like: 200 result=<digits>
# Split on '=', we want index 1
account = response.split('=', 1)[1]

if account == '-1': # digits if error
    response = agi_command('STREAM FILE astcc-account-number-invalid ""')
    response = agi_command('HANGUP')
    sys.exit(0)

if account not in ACCOUNTS: # invalid
    response = agi_command('STREAM FILE astcc-account-number-invalid ""')
    sys.exit(0)

balance = ACCOUNTS[account]['balance']

response = agi_command('STREAM FILE account-balance-is ""')
response = agi_command('SAY NUMBER %s ""' % (balance))
sys.exit(0)
```

Development Frameworks

There have been a number of efforts to create frameworks or libraries that make AGI programming easier. You will notice that several of these frameworks also appeared in Chapter 17. Just as with AMI, when evaluating a framework, we recommend you find one that meets the following criteria:

Maturity
: Has this project been around for a few years? A mature project is far less likely to have serious bugs in it.

Maintenance
: Check the age of the latest update. If the project hasn't been updated in a few years, there's a strong possibility it has been abandoned. It might still be usable, but you'll be on your own. Similarly, what does the bug tracker look like? Are there a lot of important bugs being ignored? (Be discerning here, since often the realities of maintaining a free project require disciplined triage—not everybody's features are going to get added.)

Quality of the code
: Is this a well-written framework? If it was not engineered well, you should be aware of that when deciding whether to trust your project to it.

Community
: Is there an active community of developers using this project? It's likely you'll need help; will it be available when you need it?

Documentation
: The code should be well commented, but ideally, a wiki or other official documentation to support the library is essential.

The frameworks listed in Table 18-3 met all or most of the preceding criteria at this writing. If you do not see a library listed here for your preferred programming language, it might be out there somewhere, but simply didn't make our list.

Table 18-3. AGI development frameworks

Framework	Language
Adhearsion	Ruby
Asterisk-Java	Java
AsterNET	.NET
ding-dong	Node.js
PAGI	PHP
Panoramisk	Python
StarPy	Python + Twisted

Conclusion

AGI provides a powerful interface to Asterisk that allows you to implement first-party call control in the programming language of your choice. You can take multiple approaches to implementing an AGI application. Some approaches can provide better performance, but at the cost of more complexity. AGI provides a programming environment that may make it easier to integrate Asterisk with other systems, or just provide a more comfortable call-control programming environment for the experienced programmer. In many cases, the use of a prebuilt framework will be the best approach, especially when evaluating or prototyping a complex project. For the ultimate performance, we still recommend you consider writing as much of your application as you can using the Asterisk dialplan.

CHAPTER 19
Asterisk REST Interface

People who think they know everything are a great annoyance to those of us who do.
—Isaac Asimov

The Asterisk REST Interface (ARI) was created to address the limitations inherent in developing external or enhanced functionality outside Asterisk. While AGI allows you to trigger external applications, and AMI allows you to externally supervise and control calls in progress, any attempt to integrate both into a complete external application quickly becomes complex and kludgy. ARI allows developers to build a standalone and complete application, using Asterisk as the underlying engine.

As of this writing, ARI requires a very basic dialplan in order to trigger the `Stasis()` dialplan application, which then hands the channel over to ARI. By the time you read this, it's very likely that this requirement has changed, as the Asterisk developer community has actively been working on allowing ARI to spawn without any dialplan in the middle.

Using an external interface such as ARI to control Asterisk is not necessarily going to make your life easier. The skills required to implement and troubleshoot applications of this type require a comprehensive skill set, in not only your language of choice, but also in Linux system administration, Asterisk administration, network troubleshooting, and fundamental telephony concepts. For the skilled developer, ARI can give you the power you want in your applications, but for someone learning, we recommend you consider mastering the dialplan before you dive into external development environments. The dialplan is peculiar, but it is also completely integrated, high-performance, and relatively easy to learn.

Having said that, let's get you up and running with ARI.

ARI Quick Start

This section gives you a simple working example of ARI. Later in the chapter we'll cover things in more detail.[1]

In this quick-start section we will be using a very simple HTTP access layer. You must be very careful about putting this sort of configuration into production. If, for example, you are going to run your application on a separate machine and connect it to Asterisk across a socket, you will want a more secure connection. What we're doing in this section is akin to a sailing club using dinghies to teach; useful as an introduction, but foolish and dangerous to set out to sea in such a craft.

Basic Asterisk Configuration

You should already have the Asterisk web server running, so you simply need to verify that your */etc/asterisk/http.conf* file looks similar to the following:

```
[general]
enabled = yes
bindaddr = 127.0.0.1
```

Next, a simple */etc/asterisk/ari.conf* file is needed:

```
[general]
enabled = yes
pretty = yes
[asterisk]
type = user
read_only = no
password = whateveryoudodontusethispassword
```

OK, let's load the `ari` module now:

```
$ sudo asterisk -rx 'module load res_ari.so'
Loaded res_ari.so => (Asterisk RESTful Interface)
```

Then, into our */etc/asterisk/extensions.conf* file we need an extension to trigger the `Stasis()` dialplan app:[2]

```
exten => 242,1,Noop()
  same => n,Stasis(zarniwoop)
  same => n,Hangup()
```

1 Although, to be honest, there's really nothing more to the configuration unless you're implementing one of the frameworks, which is strongly recommended if you're going to put this into a production environment, and which we will explore later.

2 We named the app "zarniwoop" because "hello-world" was used in the Digium wiki on ARI, and it seemed best to avoid overlap. You can of course name it anything you wish.

Reload your dialplan with

```
$ sudo asterisk -rx 'dialplan reload'
Dialplan reloaded.
```

At this point it might be worthwhile to simply reload Asterisk:

```
$ sudo service asterisk restart
```

There are just a few steps left, and you're ready to test your ARI environment.

Testing Your Basic ARI Environment

Since ARI depends on WebSockets, we'll need a tool to allow us to test from the command line. The Node.js package manager (`npm`) will allow us to find and install the `wscat` tool we'll use for our tests.

```
$ sudo yum -y install npm

$ sudo npm install -g wscat

/usr/bin/wscat -> /usr/lib/node_modules/wscat/bin/wscat
/usr/lib
+-- wscat@2.2.1
  +-- commander@2.15.1
  +-- read@1.0.7
  ¦ +-- mute-stream@0.0.8
  +-- ws@5.2.2
    +-- async-limiter@1.0.0
```

Now let's light it up and see what we get!

```
$ wscat -c "ws://localhost:8088/ari/events?api_key= \
  asterisk:whateveryoudodontusethispassword&app=zarniwoop"

connected (press CTRL+C to quit)
>
```

So far, so good. Let's place a call to our `Stasis()` app and see what happens.

Open up a new SSH window (leave the other one as is so you can see what happens in the `wscat` session). Connect to the Asterisk CLI in that new shell session:

```
$ sudo asterisk -rvvvv
```

Using one of your lab telephones, place a call to 242.

On the Asterisk CLI, you should see this:

```
*CLI>
  == Setting global variable 'SIPDOMAIN' to '172.29.1.57'
    -- Executing [242@sets:1] NoOp("PJSIP/SOFTPHONE_A-00000001", "") in new stack
    -- Executing [242@sets:2] Stasis("PJSIP/SOFTPHONE_A-00000001", "zarniwoop") in new stack
```

And on the `wscat` session, you should see this:

```
>
< {
  "type": "StasisStart",
  "timestamp": "2019-01-27T21:43:43.720-0500",
  "args": [],
  "channel": {
    "id": "1548643423.2",
    "name": "PJSIP/SOFTPHONE_A-00000002",
    "state": "Ring",
    "caller": {
      "name": "101",
      "number": "SOFTPHONE_A"
    },
    "connected": {
      "name": "",
      "number": ""
    },
    "accountcode": "",
    "dialplan": {
      "context": "sets",
      "exten": "242",
      "priority": 2
    },
    "creationtime": "2019-01-27T21:43:43.709-0500",
    "language": "en"
  },
  "asterisk_id": "08:00:27:27:bf:0e",
  "application": "zarniwoop"
}
>
```

OK, now we're going to open yet another shell session[3] so we can interact with this connection we've created. From this new shell, issue the following command:

```
$ curl -v -u asterisk:whateveryoudodontusethispassword -X POST \
  "http://localhost:8088/ari/channels/1548643423.2/play?media=sound:believe-its-free" sd
```

Note the `"id"` from the JSON returned on the `wscat` session must be used following the `'channels/'` portion of the `curl` command. In other words, you must match the channel identifier in your command to the channel identifier associated with your call. In this manner, you can of course wrangle many calls simultaneously.

Working with Your ARI Environment Using Swagger

Asterisk's ARI has been developed to be compatible with the OpenAPI Specification (aka Swagger), which means that many tools compatible with this spec will work with ARI. As an example, you can interact with your ARI installation using Swagger-UI, which will be useful both for debugging and as a documentation source.

3 If your computer has only one screen, now is probably the point where you're thinking what a good idea it would be to have more of them.

First up, we'll need to expose our Asterisk HTTP server to the local network (currently it's only allowing connections from 127.0.0.1). In your */etc/asterisk/http.conf* file you'll bind the HTTP server to the local IP address of your Asterisk machine:

```
$ sudo vim /etc/asterisk/http.conf

; Enable the built-in HTTP server, and only listen for connections on localhost.
[general]
enabled = yes
;bindaddr = 127.0.0.1 ; comment this out
bindaddr = 172.29.1.57 ; LAN IP OF YOUR ASTERISK SERVER
```

Next, we'll need to add a line to your */etc/asterisk/ari.conf* file:

```
$ sudo vim /etc/asterisk/ari.conf

[general]
enabled = yes
pretty = yes
allowed_origins=http://ari.asterisk.org
...
```

Save and reload the `http` and `ari` modules in Asterisk:

```
$ sudo asterisk -rx 'module reload http' ; sudo asterisk -rx 'module reload ari'
```

Now, from your development desktop, open up your browser and navigate to *http://ari.asterisk.org*.

You'll see a web page similar to Figure 19-1.

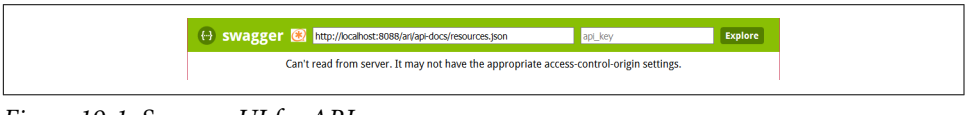

Figure 19-1. Swagger UI for ARI

Replace `localhost` with the LAN IP address of your Asterisk server, and in the api_key field, put your ARI *`user:password`* from */etc/asterisk/ari.conf* (for example, `asterisk:whateveryoudodontusethispassword`). If you've got all the configuration correct, you will be rewarded with the results in Figure 19-2.

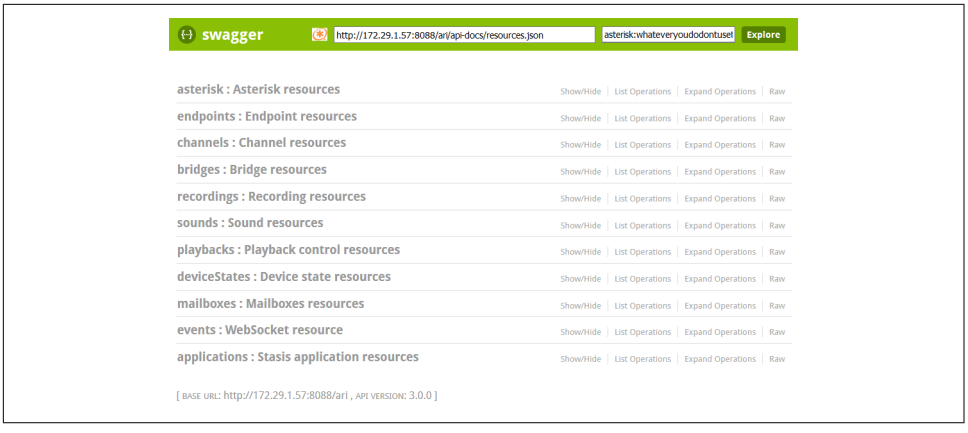

Figure 19-2. ARI Swagger

You are looking at comprehensive documentation for your ARI module, and you can actually pass queries to it as well. This is a massively useful debugging aid, and kudos to the Digium folks for it.

As an example of what this is good for, select the `endpoints:Endpoint resources` item, press the `GET` button beside `/endpoints`, and you will see the screen shown in Figure 19-3.

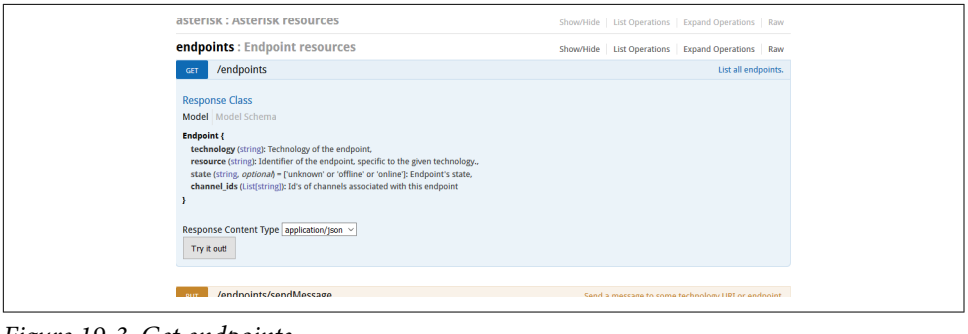

Figure 19-3. Get endpoints

Well, go ahead—press the "Try it out!" button.

Note the `"id"` of the channel in the `wscat` session, which you'll want to copy for use in the Swagger UI (you'll see several lines of JSON output relating to the call).

Perform the following actions on the channel using the Swagger UI interface: `POST: Answer` (answer the channel), `POST: hold` (place the call on hold), `DELETE: hold` (take the call off hold). Note what happens to the channel in each case.

Use of this Swagger UI is also documented over at the Asterisk wiki.

This will greatly simplify your development and testing process.

OK, that's the quick start. Let's dive in deeper to ARI.

The Building Blocks of ARI

There are three components that work together to deliver ARI:

- The RESTful interface, through which the external application communicates with Asterisk.
- A WebSocket that passes information back to the external application from Asterisk (in JSON format).
- The `Stasis()` dialplan application, which connects control of a channel to the external application.

REST

The term *RESTful* stems from Representational State Transfer (REST), which is an architectural model for web services (as opposed to, say, a protocol). The term RESTful has commonly come to refer to any API that provides interaction through URLs, with data represented in JSON format.[4] So, anything that is "RESTful" is supposed to adhere to the constraints of REST, but in practice may be implemented as a looser interpretation (which, if it gets the job done, may indeed be good enough).

WebSocket

The WebSocket connection is the mechanism that performs the communication between the internals of Asterisk and the RESTful interface. In Asterisk, events may happen that the client did not initiate, and the WebSocket allows Asterisk to signal those changes to the client.

Asterisk's built-in HTTP server potentially provides other services across a web interface. For example, WebRTC also connects through the web server. If you are making changes or adding new services, make sure you not only test the item you're working on, but also other services running through the same server, to ensure you haven't inadvertently misconfigured something else.

4 Strictly speaking, REST is far more than that, but as a practical matter, these days it seems not uncommon to assume that a REST API will be URL and JSON-based, simply because so many such services are presented in those formats.

Stasis

The Stasis Message Bus allows the core of Asterisk to communicate events with other modules and components. It is mostly internal to Asterisk; however, in the case of ARI, a dialplan application named `Stasis()` allows the dialplan to pass call control to your external ARI application.

The `Stasis()` application itself is required in order to signal to the dialplan that call control is to be passed to the external program via ARI.

> As of Asterisk 16, it is no longer necessary to write dialplan code to define a connection from an incoming channel to your ARI client application. Many developers in the Asterisk community write all their call control logic in external applications, and having to code up a few lines of dialplan just to pass channels to their app was seen as kludgy and confusing. They requested (and developed) a mechanism whereby Asterisk will create automatic dialplan to handle this function.
>
> When the API is instantiated, the application reference in the URL—for example, our app `zarniwoop`—will trigger the automatic creation of a dialplan context named according to the app name (in this case, `[stasis-zarniwoop]`), including an extension that pattern matches everything. This extension will then pass all calls arriving in that context to `Stasis(zarniwoop)`. You will need to associate your channels with the correct context (`context=stasis-zarniwoop`) in your PJSIP (or other channel) configuration tables, at which point calls to those channels will automatically be connected through `Stasis()` to the client application.
>
> If all this seems confusing, there's no reason you need to stop using actual dialplan to handle this, as we did earlier in our quick-start example.

Understanding the workings of `Stasis()` is generally not necessary unless you are going to be developing the Asterisk product itself (i.e., joining the Asterisk development team and coding new capabilities into Asterisk).

Typically, after your initial experimentation with ARI, you will want to implement a framework to help ease the work of developing your external application.

Frameworks

A production-grade application using ARI will benefit from the implementation of a framework to simplify development effort, add a layer of security, and provide a control environment.

There are several such libraries available. Which one you choose will in part be dictated by which language you prefer to use, and should also take into account whether

the framework you're interested in has an active community and is still being actively maintained.

The ones described next are listed in the Asterisk wiki. We examined the code repository for each, and while some projects are still actively maintained, others have not been updated in quite some time. If you are planning to implement one of these frameworks, you will need to do your own due diligence to ensure you can get support for it. In many cases, it may be worthwhile to reach out to the developers, and determine their consulting rates so you can ensure priority access to their time should you need it.

ari-py (and aioari) for Python

The ari-py framework was written by Digium in 2013–2014, and as of this writing had not been updated since then. This framework builds on Asterisk's Swagger.py client.

Shortly after the relase of ari-py, it was forked into the aioari project, which delivers an asynchronous version of ari-py. This code has been more steadily updated since then (although as of this writing had not been updated since early 2018). This framework should be included in your evaluation of a Python framework for ARI.

If you are looking to develop ARI applications in Python, one of these two frameworks may be what you are looking for. If you are looking to build a large ARI application, you will need to ensure that you have carefully tested the performance implications of using Python for what you are doing.

Digium has provided samples for this framework (and others) at *https://github.com/asterisk/ari-examples*.

node-ari-client

For the JavaScript folks, there is a Node.js-based ARI framework that was first released in early 2014, and as of this writing is still being updated. It is based on the automatically generated API that comes from swagger-js.

For JavaScript/Node developers, this is where you'll want to start: *https://github.com/asterisk/node-ari-client*.

Digium has provided samples for this framework (and others) at *https://github.com/asterisk/ari-examples*.

AsterNET.ARI

The Windows folks are not left out. The AsterNET.ARI project delivers a framework for .NET that augments the AsterNET project (which also includes integration with Asterisk's FastAGI and AMI interfaces).

You can find the repository for AsterNET.ARI here: *https://github.com/skrusty/AsterNET.ARI*.

Digium has provided samples for this framework (and others) at *https://github.com/asterisk/ari-examples*.

ari4java

The ari4java project is one of the most actively developed ARI frameworks we have found. It has been developed since 2013, and the repository was receiving commits at the same time as this writing.

If Java is your language, you will want to check out the ari4java repository at *https://github.com/l3nz/ari4java*.

phpari

The phpari project delivers an ARI framework for the PHP community. It has been developed since 2014, and the repository was still being updated as of this writing.

For the PHP fans, you'll find the repository at *https://github.com/greenfieldtech-nirs/phpari*.

aricpp

If you're used to writing in C++, there's even an ARI project for you. The aricpp framework consists of header files only, so you can build its functions right into whatever you're developing. This library has also been performance tested with SIPp, and while we don't have any numbers on that, it seems to us that a compiled framework that has been performance tested is very much worth taking for a spin if you have the right skills.

One of the newer of the ARI frameworks, this project benefits from regular updates. Check it out at *https://github.com/daniele77/aricpp*.

asterisk-ari-client

Yes, Ruby also has an ARI framework.

You can find it at *https://github.com/svoboda-jan/asterisk-ari*.

Conclusion

ARI provides a current-generation RESTful API that can be used to develop communications applications using popular development languages. Through it, an experienced developer can harness the power of the most successful PBX platform in history. This allows next-generation communications applications to interact with

legacy telecommunications protocols and applications, which could prove very useful as we are increasingly called to bridge the gap between past, present, and future communications technologies.

CHAPTER 20
WebRTC

The Web as I envisaged it, we have not seen it yet. The future is still so much bigger than the past.

—Tim Berners-Lee

The Browser as a Telephone

There is a new revolution brewing in internet communication, and while it isn't likely to make the news the way the open source telecom revolution did, it very definitely has the potential to quietly replace the heart of every current communication application.

Today, the internet offers a profusion of closed source conferencing applications. They all do roughly the same thing, and yet most require proprietary software to be installed before you can use them (which of course will helpfully attempt to remain loaded in the memory of your computer). Each delivers nothing much different than the last conferencing application you were forced to install (for some other meeting you've attended). Each of these companies is hoping that it will rise above the others to dominate the space. Meanwhile, WebRTC is quietly creating a standard that compellingly eliminates all concepts of proprietary multimedia communications, which hopefully will eliminate some of this narrow-minded, walled-garden thinking, and open up communications to some actual innovation.

For as long as there have been web browsers, attempts have been made to integrate multimedia into the internet experience. This has proven more difficult than expected, so that today, it is still common for the telephone to be a separate application (or, of course, a separate device altogether).

WebRTC promises to change all that.

In this chapter, we're going to get you up and running with Asterisk's interpretation of WebRTC. By no means should this be considered a comprehensive introduction; all we're going to have time to do is take you through the creation of a bog-standard video conferencing application, which is essentially the "Hello World" application that everyone uses to get started with WebRTC. It's a great way to kick the tires, but it's important to understand that WebRTC is going to be so much more.

Preliminary Knowledge

Before diving into WebRTC, there are some underlying technologies that have to come together.

First and foremost: if you're serious about getting into WebRTC, you will need access to a web developer, and ideally somebody who has a deep knowledge of the various languages, protocols, and technologies that make the internet work. WebRTC is web development, and it is bleeding-edge technology, and you are going to run into incompatibilities, browser-specific issues, undiscovered bugs, incomplete documentation, and other sorts of challenges inherent in new technology. If you are not a full-stack developer with solid networking and Linux skills, you're going to have a very steep learning curve with WebRTC!

> Probably Tsahi Levent-Levi said it best:
>
> *WebRTC is a technology that is part VoIP and part Web. ... In order to really be a professional WebRTC developer, you need to be able to grasp two very different technical domains:*
>
> 1. *You need to know how VoIP works. How media runs over the network in real time (things like RTP, RTCP, Jitter Buffer, and lots of other acronyms).*
> 2. *You need to know and understand how to develop for the web—frontend and backend (full stack developer anyone?). JavaScript is a given. Bonus points for Node.js.*

So, yeah, you'll want to be a full-stack developer, plus a VoIP guru, if you want to comfortably dive into WebRTC. We say this not to scare you off, but rather to assure you that if you find it challenging, it's not due to any shortcoming on your part, but simply because this is complex, multilayered stuff.

Having said all that, it is possible to get a taste of WebRTC without all of that, and in this chapter we're going to configure Asterisk to support WebRTC, and run a prebuilt web application that will demonstrate the basic audio/video capabilities of Asterisk's WebRTC implementation. You're still going to have that steep learning curve, but hopefully we've delivered a foundation on which to build.

Configuring Asterisk for WebRTC

To pass calls through Asterisk using WebRTC, the PJSIP channel driver must be used. The configuration will be similar to that of standard SIP telephones, but not identical.

For this we'll need a transport type, which we'll add to the */etc/asterisk/pjsip.conf* file:

```
[transport-udp]
type=transport
protocol=udp
bind=0.0.0.0

[transport-tls]
type=transport
protocol=tls
bind=0.0.0.0
cert_file=/home/asterisk/certs/self-signed.crt
priv_key_file=/home/asterisk/certs/self-signed.key
```

That's all for editing the config file. For the rest of the PJSIP changes, we'll be using the database.[1]

We're going to create two new subscribers named `WS_PHONE_A` and `WS_PHONE_B`. The WebRTC client will use the credentials for these endpoints to communicate with the PJSIP channel driver in Asterisk (i.e., to make phone calls).

Two records need to be added to the `ps_aors` table:

```
INSERT into asterisk.ps_aors
(id, max_contacts)
values ('WS_PHONE_A', 5),
       ('WS_PHONE_B', 5)
;
```

Corresponding `ps_auth` records are needed:

```
INSERT into asterisk.ps_auths
(id, auth_type, password, username)
values ('WS_PHONE_A','userpass','spiderwrench','WS_PHONE_A'),
       ('WS_PHONE_B','userpass','arachnoratchet','WS_PHONE_B')
;
```

We then create the endpoints themselves:

```
INSERT INTO asterisk.ps_endpoints
 (id,aors,auth,context,
 transport,dtls_auto_generate_cert,webrtc,disallow,allow)
VALUES
 ('WS_PHONE_A','WS_PHONE_A','WS_PHONE_A','sets',
  'transport-tls','yes','yes','all','vp8,opus,ulaw'),
 ('WS_PHONE_B','WS_PHONE_B','WS_PHONE_B','sets',
 'transport-tls','yes','yes','all','vp8,opus,ulaw');
```

1 Note that you can configure the PJSIP channel driver completely using the config file, but in this book we're only doing so where necessary, and otherwise are using the database for PJSIP channel configuration.

In Chapter 4 we already generated our certificates, so we should be able to use them here as well.

```
$ ls -l /home/asterisk/certs/
```

That should take care of the channel configuration for our WebRTC example.

We'll now need to configure Asterisk's web server to handle HTTPS.

```
$ sudo vim /etc/asterisk/http.conf

[general]
enabled=yes
bindaddr=0.0.0.0
bindport=8088
tlsenable=yes
tlsbindaddr=0.0.0.0:8089
tlscertfile=/home/asterisk/certs/self-signed.crt
tlsprivatekey=/home/asterisk/certs/self-signed.key
```

Save and restart Asterisk.

```
$ sudo service asterisk restart
```

Verify that Asterisk is now running not just an HTTP server, but also HTTPS:

```
*CLI> http show status

HTTP Server Status:
Server Enabled and Bound to 0.0.0.0:8088
HTTPS Server Enabled and Bound to 0.0.0.0:8089

Enabled URI's:
/ws => Asterisk HTTP WebSocket
```

You're looking in the output for `HTTPS` to verify that the certificates are working, and you also want to see `/ws` as that indicates the WebSockets components have loaded.

Hint: if it's not working, always check */var/log/messages* for any SELinux messages.

```
$ sudo grep sealert /var/log/messages
```

The firewall isn't currently configured for those ports, so we'll need to add a few rules to handle that:

```
$ sudo firewall-cmd --zone=public --add-port=8088/tcp
$ sudo firewall-cmd --zone=public --add-port=8088/tcp --permanent
$ sudo firewall-cmd --zone=public --add-port=8089/tcp
$ sudo firewall-cmd --zone=public --add-port=8089/tcp --permanent
$ sudo firewall-cmd --zone=public --add-port=5061/udp
$ sudo firewall-cmd --zone=public --add-port=5061/udp --permanent
```

At this point you need to fire up your web browser and make a connection. Your browser will complain about the connection if you are using a self-signed certificate,

but it will allow you to make the connection. This is a critical step, as you need to tell your browser to store the certificate permanently, so that WebRTC can use the WebSocket connection. The following URL will connect you:

```
https://ip-of-asterisk-server:8089/ws
```

If you get to an `Upgrade Required` message, *that's a good thing*. It means that the connection is good, and that's just the protocol telling you there's not enough technology being served up for this to be an actual WebSocket connection. We're where we need to be.

Of course the next thing is to actually experience a WebRTC session through this environment we've configured, and in order to test all this out, we're going to need to fire up our browser and load some sort of WebRTC client into it. The next section will do just that.

Cyber Mega Phone

In order to see WebRTC in action on your Asterisk system, you'll need something running on your browser. The easiest way to see this in action is to take Digium's *Cyber Mega Phone* for a spin. This will allow you to quickly set up a working WebRTC session using Asterisk.

First up, since WebRTC requires the use of TLS (it's not optional, as it is with SIP), we're going to nag you one more time to verify that your certificates are installed. If you haven't yet done so, now is the time to work through Chapter 4, or there is also a script provided as part of the Asterisk source code that will generate the keys and certificates (you'll find it in the Asterisk source code under the */home/astmin/src/asterisk-16.<TAB>/contrib/scripts/* folder. The script is named *ast_tls_cert*, and it is documented on the Asterisk wiki.

OK, now we need a tiny bit of dialplan for our WebRTC calls to arrive at:

```
$ vim /etc/asterisk/extensions.conf

exten => 246,1,Noop()
  same => n,Answer()
  same => n,Wait(0.5)
  same => n,StreamEcho(4)
  same => n,Hangup()
```

The Cyber Mega Phone itself is found at GitHub, under the Asterisk account.

You can download the code and run it from your local PC, or you can load it into a web server and serve it from there.

Let's serve it up from our Asterisk server:

```
$ cd /var/lib/asterisk/static-http

$ sudo git clone https://github.com/asterisk/cyber_mega_phone_2k.git
```

```
$ sudo chown -R asterisk:asterisk cyber_mega_phone_2k ; sudo chmod 755 cyber_mega_phone_2k
```

We'll need a small change to the configuration of Asterisk's HTTP server to allow it to serve static content.

```
$ sudo vim /etc/asterisk/http.conf

[general]
enabled=yes
bindaddr=0.0.0.0
bindport=8088
tlsenable=yes
tlsbindaddr=0.0.0.0:8089
tlscertfile=/home/asterisk/certs/asterisk.crt
tlsprivatekey=/home/asterisk/certs/asterisk.key
enablestatic=yes
redirect=/cmp2k /static/cyber_mega_phone_2k/index.html
```

Save and reload the `http` module from your Asterisk console:

```
*CLI> module reload http
```

Now, using your browser, you can navigate over to your new WebRTC client app:

```
https://your asterisk server:8089/cmp2k
```

If all went as planned, you should see something like Figure 20-1.

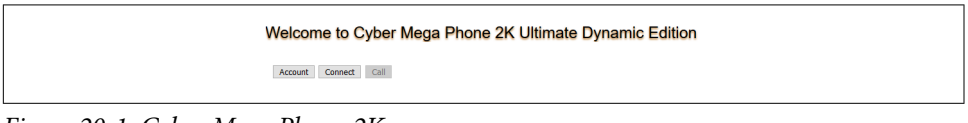

Figure 20-1. Cyber Mega Phone 2K

Press the Account button, and input the credentials for your WebRTC user (see Figure 20-2).

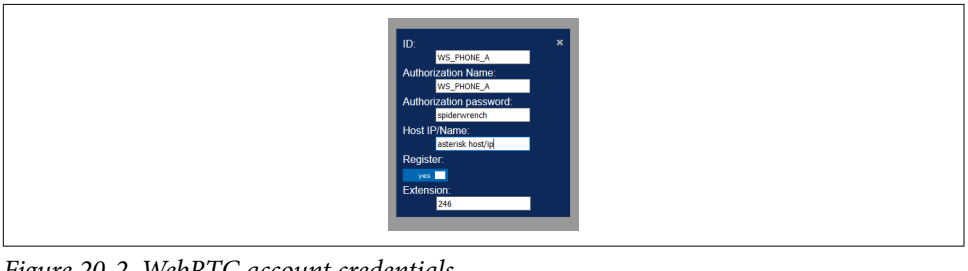

Figure 20-2. WebRTC account credentials

Once you've input the details relevant to your system, press X to save and close.

Now, you can press the Connect button, and if all went well your WebRTC client should register to Asterisk (this would be a good time to monitor the Asterisk console to see what's happening and whether there are any errors).

If you press the Call button now, you should end up connected via WebRTC, and you'll see two windows (Figure 20-3). One of them is your local video, and the other is reflected back from the far end (i.e., it is simulating another user by echoing back what you sent). If all your audio is working too, you might even get some feedback noise!

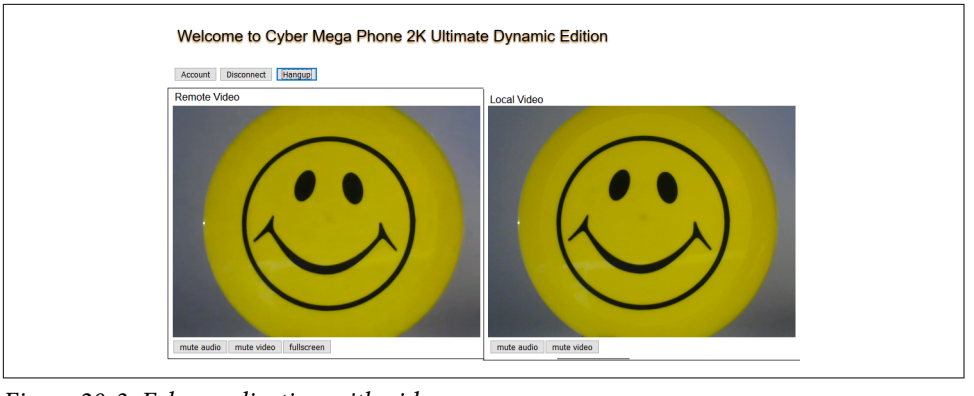

Figure 20-3. Echo application with video

You see that there is a Remote Video window alongside the Local Video window. We haven't achieved much to brag about, perhaps, but your Asterisk system is handling WebRTC, so smile and take a break. You've earned it.

More About WebRTC

The WebRTC ecosystem is rapidly evolving, and what is true as of this writing may not be true in the near future. We have found the following resources to be very helpful:

- Tsahi Levant-Levi is involved in many different WebRTC initiatives, and he generously shares knowledge relating to how to learn WebRTC. Check out his blog-geek.me website. Follow him.
- A group of folks under the Kranky Geek handle produced some WebRTC conferences, and shared many useful videos on YouTube. The YouTube channel Kranky Geek is where you'll find them.
- Get familiar with the various signaling protocols that are popular with WebRTC: SIP, VIRTO (from the FreeSwitch project), XMPP, and even JSON.
- Look up various WebRTC signaling libraries. Currently, the popular ones include: sipML5 (arguably the very first WebRTC library) and JsSIP (plus a fork of JsSIP named *SIP.js*).

- webrtc.org is the official home of WebRTC, and certainly deserves some of your time. Check out the Getting Started page.
- O'Reilly's online learning platform has a few videos that are worth a watch. For any books and videos, keep an eye on the publish date, since anything older than a year or two is likely to be out of date—WebRTC is still under rapid development.

There is so much more to learn, but we're out of pages here.

Conclusion

WebRTC is exciting and important, and it's very likely that VoIP developers and integrators are going to need to be familiar with this technology if they are to keep their skill sets relevant. As of this writing, WebRTC is still very much a work in progress. As with any exploration of new frontiers, those who blaze a trail must be creative, persistent, optimistic, and tough.

Asterisk could be a useful component in a future-ready VoIP environment, serving at least as a bridge between next-generation WebRTC products and old-school telecommunications.

CHAPTER 21
System Monitoring and Logging

Chaos is inherent in all compounded things. Strive on with diligence.
—The Buddha

Asterisk comes with several subsystems that allow you to obtain detailed information about the workings of your system. Whether for troubleshooting or for tracking usage for billing or staffing purposes, Asterisk's various monitoring modules can help you keep tabs on the inner workings of your system.

logger.conf

When troubleshooting issues in your Asterisk system, you will find it very helpful to refer to some sort of historical record of what was going on in the system at the time the reported issue occurred. The parameters for the storing of this information are defined in */etc/asterisk/logger.conf*.

Ideally, you might want the system to store a record of each and every thing it does. However, there is a cost to doing this. On a busy system, with full debug logging enabled, a large amount of data will be generated. Although storage is far cheaper today than it was when Asterisk was young, it may still be necessary to achieve a balance between detail and storage requirements.

The */etc/asterisk/logger.conf* file allows you to define all sorts of different levels of logging, to multiple files if desired. This flexibility is excellent, but it can also be confusing.

The format of an entry in the *logger.conf* file is as follows:

```
filename => type[,type[,type[,...]]]
```

We have already been working with the *logger.conf* file, so you will already have entries in it similar to the following:

```
[general]
exec_after_rotate=gzip -9 ${filename}.2;

[logfiles]
;debug => debug
;console => notice,warning,error,verbose
console => notice,warning,error,debug
messages => notice,warning,error
full => notice,warning,error,debug,verbose,dtmf,fax
;full-json => [json]debug,verbose,notice,warning,error,dtmf,fax
;syslog keyword : This special keyword logs to syslog facility
;syslog.local0 => notice,warning,error
```

If you make any changes to this file, you will need to reload the logger by issuing the following command from the shell:

```
$ sudo touch full messages
$ chown asterisk:asterisk /var/log/asterisk/*
$ asterisk -rx 'logger reload'
```

or from the Asterisk CLI:

```
*CLI> logger reload
```

Verbose Logging: Useful but Dangerous

We struggled with whether to recommend adding the following line to your *logger.conf* file:

```
verbose => notice,warning,error,verbose
```

This is quite possibly one of the most useful debugging tools you have when building and troubleshooting a dialplan, and therefore it is highly recommended. The danger comes from the fact that if you forget to disable this when you are done with your debugging, you will have left a ticking time bomb in your Asterisk system, which will slowly fill up the hard drive and kill your system one day, several months or years from now, when you are least expecting it.

Use it. It's fantastic. Just remember that you will need to manage your storage to ensure your logfiles don't fill up your drive!

You can specify any filename you want, but the special filename *console* will in fact print the output to the Asterisk CLI, and not to any file on the hard drive. All other filenames will be stored in the filesystem in the directory */var/log/asterisk*. The *logger.conf* types are outlined in Table 21-1.

Table 21-1. logger.conf types

Type	Description
`notice`	You will see a lot of these during a reload, but they will also happen during normal call flow. A notice is simply any event that Asterisk wishes to inform you of.
`warning`	A warning represents a problem that could be severe enough to affect a call (including disconnecting a call because call flow cannot continue). Warnings need to be addressed.
`error`	Errors represent significant problems in the system that must be addressed immediately.
`debug`	Debugging is only useful if you are troubleshooting a problem with the Asterisk code itself. You would not use `debug` to troubleshoot your dialplan, but you would use it if the Asterisk developers asked you to provide logs for a problem you were reporting. Do not use `debug` in production, as the amount of detail stored can fill up a hard drive in a matter of days.[a]
`verbose`	This is one of the most useful of the logging types, but it is also one of the more risky to leave unattended, due to the possibility of the output filling your hard drive.[b]
`dtmf`	Logging DTMF can be helpful if you are getting complaints that calls are not routing from the automated attendant correctly.
`fax`	This type of logging causes fax-related messages from the fax technology backend (`res_fax_spandsp` or `res_fax_digium`) to be logged to the fax logger.
`*`	This will log everything (and we mean *everything*). Do not use this unless you understand the implications of storing this amount of data. It will not end well.

[a] This is not theory. It has happened to us. It was not fun.

[b] It's not as risky as `debug`, since it'll take months to fill the hard drive, but the danger is that it will happen, say, a year later when you're on summer vacation, and it will not immediately be obvious what the problem is. Not fun.

There is a peculiarity in Asterisk's logging system that will cause you some consternation if you are unaware of it. The level of logging for the `verbose` and `debug` logging types is tied to the verbosity as set in the console. This means that if you are logging to a file with the `verbose` or `debug` type, and somebody logs into the CLI and issues the command `core set verbose 0`, or `core set debug 0`, the logging of those details to your logfile will stop.

Reviewing Asterisk Logs

Searching through logfiles can be a challenge. The trick is to be able to filter what you are seeing so that you are only presented with information that is relevant to what you are searching for.

To start with, you will need to have an approximate idea of when the trouble you are looking for occurred. Once you are oriented to the approximate time, you will need to find clues that will help you to identify the call in question. Obviously, the more information you have about the call, the faster you will be able to pin it down.

Asterisk 11 introduced a logging feature that helps with debugging a specific call. Log entries associated with a call now include a call ID. This call ID can be used with `grep`

to find all log entries associated with that call. In the following example log entry, the call ID is `C-00000004`:

```
[Dec  4 08:22:32] WARNING[14199][C-00000004]: app_voicemail.c:6286
leave_voicemail: No entry in voicemail config file for '234123452'
```

In earlier versions of Asterisk, there is another trick you can use. If, for example, you are doing verbose logging, you should note that each distinct call has a thread identifier, which, when used with `grep`, can often help you to filter out everything that does not relate to the call you are trying to debug. For example, in the following verbose log, we have more than one call in the log, and since the calls are happening at the same time, it can be very confusing to trace one call:

```
$ tail -1000 verbose
[Mar 11 …] VERBOSE[31362] logger.c:     -- IAX2/shifteight-4 answered Zap/1-1
[Mar 11 …] VERBOSE[2973] logger.c:     -- Starting simple switch on 'Zap/1-1'
[Mar 11 …] VERBOSE[31362] logger.c:   == Spawn extension (shifteight, s, 1)
exited non-zero on 'Zap/1-1'
[Mar 11 …] VERBOSE[2973] logger.c:     -- Hungup 'Zap/1-1'
[Mar 11 …] VERBOSE[3680] logger.c:     -- Starting simple switch on 'Zap/1-1'
[Mar 11 …] VERBOSE[31362] logger.c:     -- Hungup 'Zap/1-1'
```

To filter on one call specifically, we could `grep` on the thread ID. For example:

```
$ grep 31362 verbose
```

would give us:

```
[Mar 11 …] VERBOSE[31362] logger.c:     -- IAX2/shifteight-4 answered Zap/1-1
[Mar 11 …] VERBOSE[31362] logger.c:   == Spawn extension (shifteight, s, 1)
exited non-zero on 'Zap/1-1'
[Mar 11 …] VERBOSE[31362] logger.c:     -- Hungup 'Zap/1-1'
```

This method does not guarantee that you will see everything relating to one call, since a call could in theory spawn additional threads, but for basic dialplan debugging we find this approach to be very useful when the call IDs from Asterisk 11 are not available.

Logging to the Linux syslog Daemon

Linux contains a very powerful logging engine, which Asterisk can take advantage of. While a discussion of all the various flavors of `syslog` and all the possible ways to handle Asterisk logging would be beyond the scope of this book, suffice it to say that if you want to have Asterisk send logs to the `syslog` daemon, you simply need to specify the following in your */etc/asterisk/logger.conf* file:

```
syslog.local0 => notice,warning,error ; or whatever type(s) you want to log
```

You will need a designation in your *syslog* configuration file[1] named `local0`, which should look something like:

```
local0.*      /var/log/asterisk/syslog
```

You can use `local0` through `local7` for this, but check your *syslog.conf* file to ensure that nothing else is using one of those `syslog` channels.

The use of `syslog`[2] allows for much more powerful logging, but it also requires more knowledge than simply allowing Asterisk to log to files. It's mostly going to be useful if you're already collecting other logs on the system into some centralized `syslog` server.

Verifying Logging

You can view the status of all your *logger.conf* settings through the Asterisk CLI by issuing the command:

```
*CLI> logger show channels
```

You should see output similar to:

```
Channel                   Type     Status  Configuration
-------                   ----     ------  -------------
syslog.local0             Syslog   Enabled  - NOTICE WARNING ERROR VERBOSE
/var/log/asterisk/verbose File     Enabled  - NOTICE WARNING ERROR VERBOSE
/var/log/asterisk/messages File    Enabled  - NOTICE WARNING ERROR
                          Console  Enabled  - NOTICE WARNING ERROR DTMF=
```

Log Rotation

There is some log rotation support built into Asterisk. Log rotation will be done in the following cases:

- If you run the `logger rotate` Asterisk CLI command:

  ```
  *CLI> logger rotate
  ```

- During a configuration reload if any existing logfiles are greater than 1 GB in size
- If Asterisk receives the `SIGXFSZ` signal, indicating that a file it was writing to is too large

1 Which will normally be found at */etc/syslog.conf*.

2 And `rsyslog`, `syslog-ng`, and what-all-else.

Call Detail Records

The CDR system in Asterisk is used to log the history of calls in the system. In some deployments, these records are used for billing purposes. In others, call records are used for analyzing call volumes over time. They can also be used as a debugging tool by Asterisk administrators.

CDR Contents

A CDR has a number of fields that are included by default. Table 21-2 lists them.

Table 21-2. Default CDR fields

Option	Value/example	Notes
`accountcode`	`12345`	An account ID. This field is user-defined and is empty by default.
`src`	`12565551212`	The calling party's caller ID number. It is set automatically and is read-only.
`dst`	`102`	The destination extension for the call. This field is set automatically and is read-only.
`dcontext`	`PublicExtensions`	The destination context for the call. This field is set automatically and is read-only.
`clid`	`"Big Bird"` `<12565551212>`	The full caller ID, including the name, of the calling party. This field is set automatically and is read-only.
`channel`	`SIP/0004F2040808-a1bc23ef`	The calling party's channel. This field is set automatically and is read-only.
`dstchannel`	`SIP/0004F2046969-9786b0b0`	The called party's channel. This field is set automatically and is read-only.
`lastapp`	`Dial`	The last dialplan application that was executed. This field is set automatically and is read-only.
`lastdata`	`SIP/0004F2046969,30,tT`	The arguments passed to the `lastapp`. This field is set automatically and is read-only.
`start`	`2010-10-26 12:00:00`	The start time of the call. This field is set automatically and is read-only.
`answer`	`2010-10-26 12:00:15`	The answered time of the call. This field is set automatically and is read-only.
`end`	`2010-10-26 12:03:15`	The end time of the call. This field is set automatically and is read-only.
`duration`	`195`	The number of seconds between the `start` and `end` times for the call. This field is set automatically and is read-only.
`billsec`	`180`	The number of seconds between the `answer` and `end` times for the call. This field is set automatically and is read-only.

Option	Value/example	Notes
disposition	ANSWERED	An indication of what happened to the call. This may be NO ANSWER, FAILED, BUSY, ANSWERED, or UNKNOWN.
amaflags	DOCUMENTATION	The Automatic Message Accounting (AMA) flag associated with this call. This may be one of the following: OMIT, BILLING, DOCUMENTATION, or Unknown.
userfield	PerMinuteCharge:0.02	A general-purpose user field. This field is empty by default and can be set to a user-defined string.[a]
uniqueid	1288112400.1	The unique ID for the src channel. This field is set automatically and is read-only.

[a] The userfield is not as relevant now as it used to be. Custom CDR variables are a more flexible way to get custom data into CDRs.

You can access all fields of the CDR record in the Asterisk dialplan by using the CDR() function. The CDR() function is also used to set the fields of the CDR that are user-defined:

```
exten => 115,1,Verbose(Call start time: ${CDR(start)})
   same => n,Set(CDR(userfield)=zombie pancakes)
```

In addition to the fields that are always included in a CDR, it is possible to add custom fields. You do this in the dialplan by using the Set() application with the CDR() function:

```
exten => 115,1,NoOp()
   same => n,Set(CDR(mycustomfield)=coffee)
   same => n,Verbose(I need some more ${CDR(mycustomfield)})
```

If you choose to use custom CDR variables, make sure that the CDR backend that you choose is capable of logging them.

To view the built-in documentation for the CDR() function, run the following command at the Asterisk console:

```
*CLI> core show function CDR
```

In addition to the CDR() function, some dialplan applications may be used to influence CDR records. We'll look at these next.

Dialplan Applications

A few dialplan applications can be used to influence CDRs for the current call. To get a list of the CDR applications that are loaded into the current version of Asterisk, we can use the following CLI command:

```
*CLI> core show applications like CDR
    -= Matching Asterisk Applications =-
               ForkCDR: Forks the Call Data Record.
                 NoCDR: Tell Asterisk to not maintain a CDR for the current call
              ResetCDR: Resets the Call Data Record.
    -= 3 Applications Matching =-
```

Each application has documentation built into the Asterisk application, which can be viewed using the following command:

```
*CLI> core show application <application name>
```

cdr.conf

The *cdr.conf* file has a `[general]` section that contains options that apply to the entire CDR system. Additional optional sections may exist in this file that apply to specific CDR logging backend modules. Table 21-3 lists the options available in the `[general]` section.

Table 21-3. cdr.conf [general] section

Option	Value/example	Notes
`enable`	`yes`	Enable CDR logging. The default is `yes`.
`unanswered`	`no`	Log unanswered calls. Normally, only answered calls result in a CDR. Logging all call attempts can result in a large number of extra call records that most people do not care about. The default value is `no`.
`end before hexten`	`no`	Close out CDRs before running the `h` extension in the Asterisk dialplan. Normally, CDRs are not closed until the dialplan is completely finished running. The default value is `no`.
`initiated seconds`	`no`	When calculating the `billsec` field, always round up. For example, if the difference between when the call was answered and when the call ended is 1 second and 1 microsecond, `billsec` will be set to 2 seconds. This helps ensure that Asterisk's CDRs match the behavior used by telcos. The default value is `no`.
`batch`	`no`	Queue up CDRs to be logged in batches instead of logging synchronously at the end of every call. This prevents CDR logging from blocking the completion of the call teardown process within Asterisk. Using `batch` mode can be incredibly useful when working with a database that may be slow to process requests. The default value is `no`, but we recommend turning it on.[a]
`size`	`100`	Set the number of CDRs to queue up before they are logged during batch mode. The default value is `100`.
`time`	`300`	Set the maximum number of seconds that CDRs will wait in the batch queue before being logged. The CDR batch-logging process will run at the end of this time period, even if `size` has not been reached. The default value is `300` seconds.
`scheduler only`	`no`	Set whether CDR batch processing should be done by spawning a new thread, or within the context of the CDR batch scheduler. The default value is `no`, and we recommend not changing it.

Option	Value/example	Notes
safe shutdown	yes	Block Asterisk shutdown to ensure that all queued CDR records are logged. The default is yes, and we recommend leaving it that way, as this option prevents important data loss.

[a] The disadvantage of enabling this option is that if Asterisk were to crash or die for some reason, the CDR records would be lost, as they are only stored in memory while the Asterisk process exists. See safeshutdown for more information.

Backends

Asterisk CDR backend modules provide a way to log CDRs. Most CDR backends require specific configuration to get them going.

cdr_adaptive_odbc

As the name suggests, the cdr_adaptive_odbc module allows CDRs to be stored in a database through ODBC. The "adaptive" part of the name refers to the fact that it works to adapt to the table structure: there is no static table structure that must be used with this module. When the module is loaded (or reloaded), it reads the table structure. When logging CDRs, it looks for a CDR variable that matches each column name. This applies to both the built-in CDR variables and custom variables. If you want to log the built-in channel CDR variable, just create a column called channel.

Adding custom CDR content is as simple as setting it in the dialplan. For example, if we wanted to log the User-Agent that is provided by a SIP device, we could add that as a custom CDR variable:

```
exten => 105,n,Set(CDR(useragent)=${CHANNEL(useragent)})
```

To have this custom CDR variable inserted into the database by cdr_adaptive_odbc, all we have to do is create a column called useragent.

Multiple tables may be configured in the cdr_adaptive_odbc configuration file. Each goes into its own configuration section. The name of the section can be anything; the module does not use it. Here is an example of a simple table configuration:

```
[mytable]

connection = asterisk
table = asterisk_cdr
```

A more detailed example of setting up a database for logging CDRs can be found in "Storing Call Detail Records" on page 281.

Table 21-4 lists the options that can be specified in a table configuration section in the *cdr_adaptive_odbc.conf* file.

Table 21-4. cdr_adaptive_odbc.conf table configuration options

Option	Value/example	Notes
`connection`	`pgsql1`	The database connection to be used. This is a reference to the configured connection in *res_odbc.conf*. This field is required.
`table`	`asterisk_cdr`	The table name. This field is required.
`usegmtime`	`no`	Indicates whether to log timestamps using GMT instead of local time. The default value for this option is `no`.

In addition to the key/value pair fields that are shown in the previous table, *cdr_adaptive_odbc.conf* allows for a few other configuration items. The first is a column alias. Normally, CDR variables are logged to columns of the same name. An `alias` allows the variable name to be mapped to a column with a different name. The syntax is:

```
alias CDR variable => column name
```

Here is an example column mapping using the `alias` option:

```
alias src => source
```

It is also possible to specify a content filter. This allows you to specify criteria that must match for records to be inserted into the table. The syntax is:

```
filter CDR variable => content
```

Here is an example content filter:

```
filter accountcode => 123
```

Finally, *cdr_adaptive_odbc.conf* allows static content for a column to be defined. This can be useful when combined with a set of `filters`. This static content can help differentiate records that were inserted into the same table by different configuration sections. The syntax for static content is:

```
static "Static Content Goes Here" => column name
```

Here is an example of specifying static content to be inserted with CDRs:

```
static "My Content" => my_identifier
```

cdr_csv

The `cdr_csv` module is a very simple CDR backend that logs CDRs into a CSV (comma-separated values) file. The file is */var/log/asterisk/cdr-csv/Master.csv*. As long as CDR logging is enabled in *cdr.conf* and this module has been loaded, CDRs will be logged to the *Master.csv* file. We recommend that regardless of any other CDR backend you choose to configure, you leave this configured as well, as it will serve as an excellent backup should you lose other CDR data due to network or related issues.

While no options are required to get this module working, there are some options that customize its behavior. These options, listed in Table 21-5, are placed in the `[csv]` section of *cdr.conf*.

Table 21-5. cdr.conf [csv] section options

Option	Value/example	Notes
`usegmtime`	`no`	Log timestamps using GMT instead of local time. The default is `no`.
`loguniqueid`	`no`	Log the `uniqueid` CDR variable. The default is `no`.
`loguserfield`	`no`	Log the `userfield` CDR variable. The default is `no`.
`accountlogs`	`yes`	Create a separate CSV file for each different value of the `accountcode` CDR variable. The default is `yes`.

The order of CDR variables in CSV files created by the `cdr_csv` module is:

```
<accountcode>,<src>,<dst>,<dcontext>,<clid>,<channel>,<dstchannel>,<lastapp>, \
    <lastadata>,<start>,<answer>,<end>,<duration>,<billsec>,<disposition>, \
    <amaflags>[,<uniqueid>][,<userfield>]
```

Place the following lines into */etc/asterisk/cdr.conf:*

```
[general]
enable=yes

[csv]
usegmtime=yes    ; log date/time in GMT.  Default is "no"
loguniqueid=yes  ; log uniqueid.  Default is "no"
loguserfield=yes ; log user field.  Default is "no"
accountlogs=yes  ; create separate log file for each account code. Default is "yes"
;newcdrcolumns=yes ; Enable logging of post-1.8 CDR columns (peeraccount,linkedid,sequence)
                   ; Default is "no".
```

Save it, chown it, and reload the CDR module.

```
$ chown asterisk:asterisk /etc/asterisk/cdr.conf

$ sudo asterisk -rx 'module reload cdr'
```

cdr_custom

This CDR backend allows for custom formatting of CDR records in a logfile. This module is most commonly used for customized CSV output. The configuration file used for this module is */etc/asterisk/cdr_custom.conf*. A single section called `[mappings]` should exist in this file. The `[mappings]` section contains mappings between a filename and the custom template for a CDR. The template is specified using Asterisk dialplan functions.

The following example shows a sample configuration for `cdr_custom` that enables a single CDR logfile, *Master.csv*. This file will be created as */var/log/asterisk/cdr-custom/Master.csv*. The template that has been defined uses both the `CDR()` and `CSV_QUOTE()` dialplan functions. The `CDR()` function retrieves values from the CDR being logged.

The `CSV_QUOTE()` function ensures that the values are properly escaped for the CSV file format:

```
[mappings]

Master.csv => ${CSV_QUOTE(${CDR(clid)})},${CSV_QUOTE(${CDR(src)})},
   ${CSV_QUOTE(${CDR(dst)})},${CSV_QUOTE(${CDR(dcontext)})},
   ${CSV_QUOTE(${CDR(channel)})},${CSV_QUOTE(${CDR(dstchannel)})},
   ${CSV_QUOTE(${CDR(lastapp)})},${CSV_QUOTE(${CDR(lastdata)})},
   ${CSV_QUOTE(${CDR(start)})},${CSV_QUOTE(${CDR(answer)})},
   ${CSV_QUOTE(${CDR(end)})},${CSV_QUOTE(${CDR(duration)})},
   ${CSV_QUOTE(${CDR(billsec)})},${CSV_QUOTE(${CDR(disposition)})},
   ${CSV_QUOTE(${CDR(amaflags)})},${CSV_QUOTE(${CDR(accountcode)})},
   ${CSV_QUOTE(${CDR(uniqueid)})},${CSV_QUOTE(${CDR(userfield)})}
```

In the actual configuration file, the value in the *Master.csv* mapping should be on a single line.

cdr_manager

The `cdr_manager` backend emits CDRs as events on the Asterisk Manager Interface (AMI), which we discussed in detail in Chapter 17. This module is configured in the */etc/asterisk/cdr_manager.conf* file. The first section in this file is the `[general]` section, which contains a single option to enable this module (the default value is `no`):

```
[general]

enabled = yes
```

The other section in *cdr_manager.conf* is the `[mappings]` section. This allows for adding custom CDR variables to the manager event. The syntax is:

```
CDR variable => Header name
```

Here is an example of adding two custom CDR variables:

```
[mappings]

rate => Rate
carrier => Carrier
```

With this configuration in place, CDR records will appear as events on the manager interface. To generate an example manager event, we will use the following dialplan example:

```
exten => 110,1,Answer()
    same => n,Set(CDR(rate)=0.02)
    same => n,Set(CDR(carrier)=BS&S)
    same => n,Hangup()
```

This is the command used to execute this extension and generate a sample manager event:

```
*CLI> console dial 110@testing
```

Finally, this is an example manager event produced as a result of this test call:

```
Event: Cdr
Privilege: cdr,all
AccountCode:
Source:
Destination: 110
DestinationContext: testing
CallerID:
Channel: Console/dsp
DestinationChannel:
LastApplication: Hangup
LastData:
StartTime: 2010-08-23 08:27:21
AnswerTime: 2010-08-23 08:27:21
EndTime: 2010-08-23 08:27:21
Duration: 0
BillableSeconds: 0
Disposition: ANSWERED
AMAFlags: DOCUMENTATION
UniqueID: 1282570041.3
UserField:
Rate: 0.02
Carrier: BS&S
```

cdr_odbc

This module enables the legacy ODBC interface for CDR logging. New installations should use `cdr_adaptive_odbc` instead.

cdr_sqlite

This module allows posting of CDRs to an SQLite database using SQLite version 2. Unless you have a specific need for SQLite version 2 as opposed to version 3, we recommend that all new installations use `cdr_sqlite3_custom`.

This module requires no configuration to work. If the module has been compiled and loaded into Asterisk, it will insert CDRs into a table called `cdr` in a database located at */var/log/asterisk/cdr.db*.

cdr_sqlite3_custom

This CDR backend inserts CDRs into an SQLite database using SQLite version 3. The database created by this module lives at */var/log/asterisk/master.db*. This module requires a configuration file, */etc/asterisk/cdr_sqlite3_custom.conf*. The configuration file identifies the table name, as well as customizes which CDR variables will be inserted into the database.

cdr_syslog

This module allows logging of CDRs using `syslog`. To enable this, first add an entry to the system's *syslog* configuration file, */etc/syslog.conf*. For example:

```
local4.*      /var/log/asterisk/asterisk-cdr.log
```

The Asterisk module has a configuration file as well. Add the following section to */etc/asterisk/cdr_syslog.conf*:

```
[cdr]

facility = local4
priority = info
template = "We received a call from ${CDR(src)}"
```

Here is an example *syslog* entry using this configuration:

```
$ cat /var/log/asterisk/asterisk-cdr.log

Aug 12 19:17:36 pbx cdr: "We received a call from 2565551212"
```

Example Call Detail Records

We will use the `cdr_custom` module to illustrate some example CDR records for different call scenarios. The configuration used for */etc/asterisk/cdr_custom.conf* is shown in "cdr_custom" on page 361.

Single-party call

In this example, we'll show what a CDR looks like for a simple one-party call:

```
exten => 227,1,VoiceMailMain(@${GLOBAL(VOICEMAIL_CONTEXT)})
```

This is the CDR from */var/log/asterisk/cdr-custom/Master.csv* that was created as a result of calling this extension:

```
"","SOFTPHONE_A","227","sets","""101"" <SOFTPHONE_A>","PJSIP/SOFTPHONE_A-00000002",
"","Playback","hear-odd-noise",
"2019-03-04 02:31:39","2019-03-04 02:31:39","2019-03-04 02:31:42",
3,3,"ANSWERED","DOCUMENTATION","1551666699.4",""
```

Open it up in a spreadsheet and it'll be lined up neatly.

Caveats

The CDR system in Asterisk works very well for fairly simple call scenarios. However, as call scenarios get more complicated—involving calls to multiple parties, transfers, parking, and other such features—the CDR system starts to fall short. Many users report that the records do not show all the information that they expect. Many bug fixes have been made to address some of the issues, but the cost of regressions or changes in behavior when making changes in this area is very high, since these records are used for billing.

As a result, the Asterisk development team has become increasingly resistant to making additional changes to the CDR system. Instead, a new system, channel event logging (CEL), has been developed that is intended to help address logging of more complex call scenarios. Bear in mind that call detail records are simpler and easier to consume, though, so we still recommend using CDRs if they suit your needs.

Channel Event Logging

Channel event logging (CEL) provides a more flexible means of logging the details of complex call scenarios. Instead of collapsing a call down to a single log entry, a series of events are logged for the call. This provides a more accurate picture of what has happened to the call, at the expense of a more complex log.

For more details on CEL, check out the Asterisk wiki.

Conclusion

Asterisk is very good at allowing you to keep track of many different facets of its operation, from simple call detail records to full debugging of the running code. Take a look in the source code directories, and you'll find many more components than we've had space to cover here. These various mechanisms will help you in your efforts to manage your Asterisk PBX, and they represent one of the ways that Asterisk is vastly superior to most (if not all) traditional PBXs.

CHAPTER 22
Security

We spend our time searching for security and hate it when we get it.
—John Steinbeck

Security for your Asterisk system is critical, especially if the system is exposed to the internet. There is a lot of money to be made by attackers in exploiting systems to make free phone calls. This chapter provides advice on how to provide stronger security for your VoIP deployment.

Scanning for Valid Accounts

If you expose your Asterisk system to the public internet, one of the things you will almost certainly see is a scan for valid accounts. Example 22-1 contains log entries from one of the authors' production Asterisk systems.[1] This scan began with checking various common usernames, then later went on to scan for numbered accounts. It is common for people to name SIP accounts the same as extensions on the PBX. This scan takes advantage of that fact.

Use non-numeric usernames for your VoIP accounts to make them harder to guess. For example, in this book we use the MAC address of a SIP phone as its account name in Asterisk.

1 The real IP address has been replaced with 127.0.0.1 in the log entries.

Example 22-1. Log excerpts from account scanning

```
[Aug 22 15:17:15] NOTICE[25690] chan_sip.c: Registration from
'"123"<sip:123@127.0.0.1>' failed for '203.86.167.220:5061' - No matching peer
found
[Aug 22 15:17:15] NOTICE[25690] chan_sip.c: Registration from
'"1234"<sip:1234@127.0.0.1>' failed for '203.86.167.220:5061' - No matching peer
found
[Aug 22 15:17:15] NOTICE[25690] chan_sip.c: Registration from
'"12345"<sip:12345@127.0.0.1>' failed for '203.86.167.220:5061' - No matching peer
found

...

[Aug 22 15:17:17] NOTICE[25690] chan_sip.c: Registration from
'"100"<sip:100@127.0.0.1>' failed for '203.86.167.220:5061' - No matching peer found
[Aug 22 15:17:17] NOTICE[25690] chan_sip.c: Registration from
'"101"<sip:101@127.0.0.1>' failed for '203.86.167.220:5061' - No matching peer found
```

The logs on any system will be full of intrusion attempts. This is simply the nature of connecting systems to the internet. In this chapter, we will discuss some of the ways to configure your system so that it will have robust mechanisms to deal with these things.

Authentication Weaknesses

The first section of this chapter discussed scanning for usernames. Even if you have usernames that are difficult to guess, it is critical that you have strong passwords as well. If an attacker is able to obtain a valid username, they will likely attempt to brute-force the password. Strong passwords make this much more difficult.

The default authentication scheme of the SIP protocol is weak. Authentication is done using an MD5 challenge-and-response mechanism. If an attacker is able to capture any call traffic, such as a SIP call made from a laptop on an open wireless network, it will be much easier to work on brute-forcing the password, since it will not require authentication requests to the server.

Use strong passwords. There are countless resources available on the internet that help define what constitutes a strong password. There are also many strong password generators available. Use them!

Fail2ban

The previous two sections discussed attacks involving scanning for valid usernames and brute-forcing passwords. Fail2ban is an application that can watch your Asterisk logs and update firewall rules to block the source of an attack in response to too many failed authentication attempts.

Use Fail2ban when exposing Voice over IP services on untrusted networks. It will automatically update the firewall rules to block the sources of attacks.

Installation

Fail2ban is available as a package in many distributions. Alternatively, you can install it from source by downloading it from the Fail2ban website. To install Fail2ban on RHEL, you must have the EPEL repository enabled (which was handled during Chapter 3). You can install Fail2ban by running the following command:

```
$ sudo yum install fail2ban
```

The installation of Fail2ban from a package will include a startup script to ensure that it runs when the machine boots up. If you install from source, make sure that you take the necessary steps to ensure that Fail2ban is always running.

Configuration

First up, we'll want to configure the security log in Asterisk, which Fail2ban is able to make use of.

```
$ sudo vim /etc/asterisk/logger.conf
```

Uncomment the (or add a) line that reads `security => security`, and edit the `date format` so Fail2ban understands the logfile.

```
[general]
exec_after_rotate=gzip -9 ${filename}.2;
dateformat = %F %T
[logfiles]
;debug => debug
security => security
;console => notice,warning,error,verbose
console => notice,warning,error,debug
messages => notice,warning,error
full => notice,warning,error,debug,verbose,dtmf,fax
```

Then reload the Asterisk logger:

```
$ sudo asterisk -rx 'logger reload'
```

Since current versions of Fail2ban already come with an Asterisk jail definition, all we need to do is enable it:

The current best practice is to create a file */etc/fail2ban/jail.local* for this purpose (technically you can put it in */etc/fail2ban/jail.conf*, but this is more likely to be overwritten):

```
$ sudo vim /etc/fail2ban/jail.local

[asterisk]
enabled  = true
filter   = asterisk
action   = iptables-allports[name=ASTERISK, protocol=all]
           sendmail[name=ASTERISK, dest=me@shifteight.org, sender=fail2ban@shifteight.org]
logpath  = /var/log/asterisk/messages
           /var/log/asterisk/security
maxretry = 5
findtime = 21600
bantime = 86400
```

We've set up the ban for 24 hours, but you can do longer or shorter times as well if you prefer (the `bantime` is defined in seconds, so calculate accordingly). Since most attacking hosts move on after a few hours, there's no harm in unblocking an IP after 24 hours. If the host attacks again, they'll be blocked again.

Oh, you might also want to tell it to ignore your IP (or any other IP addresses that are OK to receive connection attempts from). If you haven't yet accidentally gotten yourself blocked because you were doing some lab work and misregistering, don't worry, you will eventually do this to yourself (unless, of course, you create an ignore list for appropriate IPs).

```
[DEFAULT]
ignoreip = <ip address(es), separated by commas>

[asterisk]
enabled  = true
filter   = asterisk
action   = iptables-allports[name=ASTERISK, protocol=all]
           sendmail[name=ASTERISK, dest=me@shifteight.org, sender=fail2ban@shifteight.org]
logpath  = /var/log/asterisk/messages
           /var/log/asterisk/security
maxretry = 5
findtime = 21600
bantime = 86400
```

Restart Fail2ban and you're good to go.

```
$ sudo systemctl reload fail2ban
```

Test it out if you can, from an IP address you don't mind being blocked (for example, an extra computer in your lab that can be the test subject for this). Attempt to register using bad credentials, and after five attempts (or whatever you set `maxretry` to), that IP should be blocked.

You can see what addresses the Asterisk jail is blocking with the command:

```
$ sudo fail2ban-client status asterisk
```

And if you want to unblock an IP,[2] the following command should do so.

```
$ sudo fail2ban-client set asterisk unbanip ip to unban
```

More information about Fail2ban can be found at the Fail2ban wiki.

Encrypted Media

While we gave examples in this book that used encryption, be aware that you can configure SIP so that media will be sent unencrypted. In that case, anyone intercepting the RTP traffic between two SIP peers will be able to use fairly simple tools to extract the audio from those calls.

Dialplan Vulnerabilities

The Asterisk dialplan is another area where taking security into consideration is critical. The dialplan can be broken down into multiple contexts to provide access control to extensions. For example, you may want to allow your office phones to make calls out through your service provider. However, you do not want to allow anonymous callers that come into your main company menu to be able to then dial out through your service provider. Use contexts to ensure that only the callers you intend have access to services that cost you money.

Build dialplan contexts with great care. Also, avoid putting any extensions that could cost you money in the `[default]` context.

One of the more recent Asterisk dialplan vulnerabilities to have been discovered and published is the idea of dialplan injection. A dialplan injection vulnerability begins with an extension that has a pattern that ends with the match-all character, a period. Take this extension as an example:

```
exten => _X.,1,Dial(PJSIP/otherserver/${EXTEN},30)
```

The pattern for this extension matches all extensions (of any length) that begin with a digit. Patterns like this are pretty common and convenient. The extension then sends this call over to another server using the IAX2 protocol, with a dial timeout of 30 seconds. Note the usage of the `${EXTEN}` variable here. That's where the vulnerability exists.

2 For example, yourself, because you forgot to define `ignoreip`...

In the world of Voice over IP, there is no reason that a dialed extension must be numeric. In fact, it is quite common using SIP to be able to dial someone by name. Since it is possible for non-numeric characters to be a part of a dialed extension, what would happen if someone sent a call to this extension?

```
1234&DAHDI/g1/12565551212
```

A call like this is an attempt at exploiting a dialplan injection vulnerability. In the previous extension definition, once `${EXTEN}` has been evaluated, the actual `Dial()` statement that will be executed is:

```
exten => _X.,1,Dial(PJSIP/otherserver/1234&DAHDI/g1/12565551212,30)
```

If the system has a PRI configured, this call will cause a call to go out on the PRI to a number chosen by the attacker, even though you did not explicitly grant access to the PRI to that caller. This problem can quickly cost you a whole lot of money.

There are several approaches to avoiding this problem. The first and easiest approach is to always use strict pattern matching. If you know the length of extensions you are expecting and expect only numeric extensions, use a strict numeric pattern match. For example, this would work if you are expecting four-digit numeric extensions only:

```
exten => _XXXX,1,Dial(PJSIP/otherserver/${EXTEN},30)
```

Another approach to mitigating dialplan injection vulnerabilities is by using the `FILTER()` dialplan function. Perhaps you would like to allow numeric extensions of any length. `FILTER()` makes that easy to achieve safely:

```
exten => _X.,1,Set(SAFE_EXTEN=${FILTER(0-9A-F,${EXTEN})})
    same => n,Dial(PJSIP/otherserver/${SAFE_EXTEN},30)
```

For more information about the syntax for the `FILTER()` dialplan function, see the output of the `core show function FILTER` command at the Asterisk CLI.

A more comprehensive (but also complex) approach might be to have all dialed digits validated by functions outside of your dialplan (for example, database queries that validate the dialed string against user permissions, routing patterns, restriction tables, and so forth). This is a powerful concept, but beyond the scope of this book.

Be wary of dialplan injection vulnerabilities. Use strict pattern matching or use the `FILTER()` dialplan function to avoid these problems.

Securing Asterisk Network APIs

To secure AGI, AMI, and ARI, you will need to carefully consider the following recommended practices:

- Only allow connections directly to the API from `localhost/127.0.0.1`.
- Use an appropriate framework in between the Asterisk API and your client application, and handle connection security through the framework.
- Control access to the framework and the system through strict firewall rules.

Beyond that, the same sort of security rules and best practices apply that you would follow in any mission-critical web application.

Other Risk Mitigation

There are other useful features in Asterisk that can be used to mitigate the risk of attacks. The first is to use the `permit` and `deny` options to build access control lists (ACLs) for privileged accounts. Consider a PBX that has SIP phones on a local network, but also accepts SIP calls from the public internet. Calls coming in over the internet are only granted access to the main company menu, while local SIP phones have the ability to make outbound calls that cost you money. In this case, it is a very good idea to set ACLs to ensure that only devices on your local network can use the accounts for the phones.

In your `ps_endpoints` table, the `permit` and `deny` options allow you to specify IP addresses, but you can also point to a label in the */etc/asterisk/acl.conf* file. In fact, ACLs are accepted almost everywhere that connections to IP services are configured. For example, another useful place for ACLs is in */etc/asterisk/manager.conf*, to restrict AMI accounts to the single host that is supposed to be using the manager interface.

ACLs can be defined in */etc/asterisk/acl.conf*.

```
[named_acl_1]
deny=0.0.0.0/0.0.0.0
permit=10.1.1.50
permit=10.1.1.55

[named_acl_2] ; Named ACLs support IPv6, as well.
deny=::
permit=::1/128

[local_phones]
deny=0.0.0.0/0.0.0.0
permit=192.168.0.0/255.255.0.0
```

Once named ACLs have been defined in *acl.conf*, have Asterisk load them using the `reload acl` command. Once loaded, they should be available via the Asterisk CLI:

```
*CLI> module reload acl

*CLI> acl show

acl
---
named_acl_1
named_acl_2
local_phones

*CLI> acl show named_acl_1

ACL: named_acl_1
-----------------------------------------------
  0:  deny - 0.0.0.0/0.0.0.0
  1: allow - 10.1.1.50/255.255.255.255
  2: allow - 10.1.1.55/255.255.255.255
```

Now, instead of having to potentially repeat the same `permit` and `deny` entries in multiple places, you can apply an ACL by its name. You will find an `acl` field in the `ps_endpoints` table, which you can use to point to a named ACL in the *acl.conf* file.

```
mysql> select id,transport,aors,context,disallow,allow,acl from ps_endpoints;

|id          |transport    |aors        |context|disallow|allow        |acl |
|0000f30A0A01|transport-udp|0000f30A0A01|sets   |all     |ulaw         |NULL|
|0000f30B0B02|transport-udp|0000f30B0B02|sets   |all     |ulaw         |NULL|
|SOFTPHONE_A |transport-udp|SOFTPHONE_A |sets   |all     |ulaw,h264,vp8|NULL|
|SOFTPHONE_B |transport-udp|SOFTPHONE_B |sets   |all     |ulaw,h264,vp8|NULL|

mysql> update ps_endpoints
        set acl='local_phones'
        where id in ('0000f30A0A01','0000f30B0B02','SOFTPHONE_A','SOFTPHONE_B')
       ;

mysql> select id,transport,aors,context,disallow,allow,acl from ps_endpoints;

|id          |transport    |aors        |context|disallow|allow        |acl         |
|0000f30A0A01|transport-udp|0000f30A0A01|sets   |all     |ulaw         |local_phones|
|0000f30B0B02|transport-udp|0000f30B0B02|sets   |all     |ulaw         |local_phones|
|SOFTPHONE_A |transport-udp|SOFTPHONE_A |sets   |all     |ulaw,h264,vp8|local_phones|
|SOFTPHONE_B |transport-udp|SOFTPHONE_B |sets   |all     |ulaw,h264,vp8|local_phones|
```

Use ACLs when possible on all privileged accounts for network services.

Another way you can mitigate security risk is by configuring call limits. The recommended method for implementing call limits is to use the `GROUP()` and `GROUP_COUNT()` dialplan functions. Here is an example that limits the number of calls from each SIP peer to no more than two at a time:

```
exten => _X.,1,Set(GROUP(users)=${CHANNEL(endpoint)})
    same => n,NoOp(${CHANNEL(endpoint)} : ${GROUP_COUNT(${CHANNEL(endpoint)})} calls)
    same => n,GotoIf($[${GROUP_COUNT(${CHANNEL(endpoint)})} > 2]?denied:continue)

    same => n(denied),NoOp(There are too many calls up already.  Hang up.)
    same => n,HangUp()

    same => n(continue),NoOp(continue processing call as normal here ...)
```

Use call limits to ensure that if an account is compromised, it cannot be used to make hundreds of phone calls at a time.

Resources

Some security vulnerabilities require modifications to the Asterisk source code to resolve. When those issues are discovered, the Asterisk development team puts out new releases that contain only fixes for the security issues, to allow for quick and easy upgrades. When this occurs, the Asterisk development team also publishes a security advisory document that discusses the details of the vulnerability. We recommend that you subscribe to the *asterisk-announce* mailing list to make sure that you know about these issues when they come up.

Subscribe to the asterisk-announce list to stay up to date on Asterisk security vulnerabilities.

One of the most popular tools for SIP account scanning and password cracking is SIPVicious. We strongly encourage that you take a look at it and use it to audit your own systems. If your system is exposed to the internet, others will likely run SIPVicious against it, so make sure that you do that first.

Conclusion—A Better Idiot

There is a maxim in the technology industry that states, "As soon as something is made idiot-proof, nature will invent a better idiot." The point of this statement is that no development effort can be considered complete. There is always room for improvement.

When it comes to security, you must always bear in mind that the people who are looking to take advantage of your system are highly motivated. No matter how secure your system is, somebody will always be looking to crack it.

We're not advocating paranoia, but we are suggesting that what we have written here is by no means the final word on VoIP security. While we have tried to be as comprehensive as we can be in this book, you must accept responsibility for the security of your system.

The criminals are working hard to find weaknesses and exploit them.

CHAPTER 23
Asterisk: A Future for Telephony

Hey, I just met you,
And this is crazy,
But here's my number,
So call me, maybe?
—Carly Rae Jepsen

We have arrived at the final chapter of this book. We've covered a lot (and this book has been massively modified over the years), but we hope that we have made it clear that we've merely scratched the surface of Asterisk. To wrap things up, we want to spend some time exploring what we might see from Asterisk and open source telephony in the near future.

When we wrote the first edition of *Asterisk: The Future of Telephony*, we confidently asserted that open source communications engines such as Asterisk would cause a shift in thinking that would transform the telecommunications industry. In many ways, our belief has been proven correct; however, some might argue it was a hollow victory, because what has also happened during that time is a shift away from telecommunications as a primary real-time communications medium. Younger generations have little or no use for telephone calls, and consider them disruptive, annoying, and in some cases even rude.

So, even as Asterisk ushered in a transformed age for the telecommunications industry, it has now become the standard-bearer for technologies that many say are as good as dead.

While there can be no doubt that the telephone is no longer the primary communications technology in the world (not by a long shot!), when we distill communications down to their essence, we find there may be a future for this stuff yet.

The Telephone Is Dead (Except When It's Not)

While it is obvious that younger generations do not use the telephone much anymore, it is also true that older generations are very frustrated and disillusioned by modern communications technologies. For them, the telephone represents a reliable, predictable, and easy-to-understand communications method, and they are likely to continue using it for the remainder of their lives. Since there are an awful lot of old people in this world, and many of them are senior executives, decision makers, and shareholders—not to mention well-heeled customers—it seems to be a good strategy for businesses today to continue to ensure that their customers can reach them through the telephone.

When one has tried all other methods of communication, such as email, webforms, and perhaps even text messaging, one will finally pick up the phone and call. It seems that in many cases, a problem that could not get sorted out any other way is finally resolved over the phone.

It would also be correct to say that the increasingly poor job companies are doing in handling communications with their customers is a source of much frustration and confusion. However, as always, where there is a problem, there exists opportunity. Companies that retain a commitment to an excellent telecom infrastructure may find themselves with a distinct competitive advantage, using nothing more complicated than good old-fashioned customer service. If you wish to service customers over age 50, you would do well to keep your telephone system running well.

Another interesting component of traditional telecommunications networks is that while we can never be sure that we are using the same conferencing software as each other (never in history have so many near-identical apps had to be installed just to allow people to talk to each other), we can be reasonably sure that if one of us picks up the phone and dials the other's phone number, a successful conversation will be possible, without any troubleshooting or software installation. In an age where it seems no conference call can start without someone having to troubleshoot their app, this kind of universal consistency and reliability likely still has some value. Today's hot new office collaboration software is tomorrow's forgotten toy (whither thou, Skype?). The brave old telephone soldiers on.

We're not sure the telephone is dead just yet.

Communications Overload

In many ways, the ability to communicate defines our species. Yes, other critters are able to signal each other in basic ways, but our fascination with creating ever-changing and innovative ways to connect to each other is not something we've encountered in any other being.

From the carrier pigeon to the postal service to the telegraph, telephone, and television, each new technology served the same goal: improving our ability to communicate. Today, we have achieved a most remarkable thing: it is now reasonable to expect instant communication with almost anyone on the planet.

The challenge we never predicted is that too much of a good thing has begun to overwhelm us. It will be interesting to see how this plays out culturally.

The Problems with Open Source Development

Although Alexander Graham Bell is most famously remembered as the father of the telephone,[1] the reality is that during the latter half of the 1800s, dozens of minds were working toward the goal of carrying voice over telegraph lines. These people were mostly business-minded folks, looking to create a product through which they might make their fortunes.

We have come to think of traditional telephone companies as monopolies, but this was not true in their early days. The early history of telephone service took place in a very competitive environment, with new companies springing up all over the world, often with little or no respect for the patents they might be violating. Many famous monopolies got their start through the waging (and winning) of patent wars.

It's interesting to contrast the history of the telephone with the history of GNU Linux and the internet. While the telephone was created as a commercial exercise, and the telecom industry was forged through lawsuits and corporate takeovers, Linux and the internet arose out of the academic community, which has tended to value the sharing of knowledge over profit.

Unfortunately, once again too much of a good thing has begun to overwhelm. What we have seen recently is a loss of vision for open source development. Too few developers have gotten tired of the demands of too many users unwilling to contribute. Most open source projects have—out of necessity—had to shield the development team from the selfish demands of those who intend to only take, and never give. This abuse of the developers has, sadly, even extended to companies that have built highly profitable businesses on open source projects that they have never contributed a dime to. Multibillion-dollar businesses, profiting from the efforts of a team barely able to pay their bills, is not a sustainable development model. It remains to be seen how this story will play out, but open source software is not what it was 10 years ago.

Asterisk is fortunate in that it is funded by the efforts of Sangoma/Digium, the parents of the project. Their challenge has and always will be to figure out how to nurture the product in such a way that the requirements of the business are compati-

1 Ever heard of Elisha Gray or Antonio Meucci?

ble with the needs of the project. Not an easy task. We'll be cheering for them. They have done a remarkable job thus far.

The Future of Asterisk

So, does Asterisk have a future? We don't see why it shouldn't. It continues to do what it has always done, and it also works hard to be compatible with suitable technologies coming down the pipe. If nothing else, Asterisk will continue to be very good at integrating with telephone technologies, and we're not prepared to call that story fully told yet.

WebRTC

Keep an eye on WebRTC. We suspect that if open source and open-standards communications has any sort of future, WebRTC stands as the most promising candidate to achieve that.

Asterisk is not likely to be at the center of that revolution, but it will have a role to play.

The Future of Telephony

Telephony may look dead, but we still see movement in the tail, and it's a long tail indeed.

Index

Symbols

A

B

C

D

E

Q

R

S

T

U

V

W

Z

About the Authors

Jim Van Meggelen is a founding partner and CTO of Clearly Core Inc., a Canada-based provider of open source telephony solutions. He has nearly 30 years of enterprise telecom experience, with extensive knowledge of both legacy telecom and VoIP.

Russell Bryant is a Distinguished Engineer at Red Hat, where he works on cloud infrastructure projects. Prior to working for Red Hat, Russell spent seven years working for Digium on the Asterisk project. Russell's role at Digium began as a software developer and concluded with being the leader of the Asterisk project and engineering manager for the team focused on Asterisk development.

Leif Madsen is the Cloud Service Assurance Architect within the CloudOps team at Red Hat, where he leads the engineering effort to provide Service Assurance to both telecommunications and enterprise companies. He first got involved with the Asterisk community when he was looking for a voice-conferencing solution. Once he learned that there was no official Asterisk documentation, he cofounded the Asterisk Documentation Project.

Colophon

The animals on the cover of *Asterisk: The Definitive Guide* are starfish (*Asteroidea*), a group of echinoderms (spiny-skinned invertebrates found only in the sea). Most starfish have fivefold radial symmetry (arms or rays branching from a central body disc in multiples of five), though some species have four or nine arms. There are over 1,500 species of starfish.

Starfish live on the sea floor and in tidal pools, clinging to rocks and moving (slowly) using a water-based vascular system to manipulate hundreds of tiny, tube-like legs, called *podia*. A small bulb or *ampulla* at the top of the tube contracts, expelling water and expanding the starfish's leg. The ampulla relaxes, and the leg retracts. At the tip of each leg is a suction cup that allows the starfish to pry open clam, oyster, or mussel shells. Starfish are carnivores; they eat coral, fish, bivalves, and snails.

Starfish can flex and manipulate their arms to fit into small places. At the end of each arm is an eyespot, a primitive sensor that detects light and helps the starfish determine direction. Starfish also have the ability to regenerate a missing limb. Some species can even regrow a complete, new starfish from a severed arm.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. The cover illustration is by Karen Montgomery, based on a black and white engraving from the *Dover Pictorial Archive*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.