Exim

The Mail Transfer Agent

Exim

The Mail Transfer Agent

Philip Hazel

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

Exim: The Mail Transfer Agent
by Philip Hazel

Published by O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472.

Editor: Andy Oram

Production Editor: Mary Brady

Cover Designer: Ellie Volckhausen

Printing History:

June 2001: First Edition.

Library of Congress Cataloging-in-Publication Data

Hazel, Philip
Exim: the mail transfer agent/by Philip Hazel p.cm.
ISBN 0-596-00098-7
1. Exim (Computer program) 2. Email--Computer programs I. Title

TK5105.73 .H39 2001
004.692--dc21 2001036079

[DS]

Table of Contents

Preface

Back in 1995, the central computing services at Cambridge University were running a variety of mail transfer agents, including Sendmail, Smail 3, and PP. Some years before, I had converted the systems whose mail I managed from Sendmail to Smail to make it easier to handle the special requirements of the early 1990s in UK academic networking during the transition from a private X.25-based network to the Internet. By 1995, the transition was complete, and it was time to move on.

Up to that time, the Internet had been a pretty friendly place, and there was little need to take many precautions against hostile acts. Most sites ran open mail relays, for example. It was clear, however, that this situation was changing and that new requirements were arising. I had done some modifications to the code of Smail, but by then it was eight-year-old code, written in prestandard C, and originally designed for use in a very different environment that involved a lot of support for UUCP. I therefore decided to see if I could build a new MTA from scratch, taking the basic philosophy of Smail and extending it, but leaving out the UUCP support, which was not needed in our environment. Because I wasn't exactly sure what the outcome would be, I called it *EXperimental Internet Mailer* (Exim).

One of my colleagues in Computer Science got wind of what I was doing, begged for an evaluation copy, and promptly put it into service, even before I was running it on my hosts. He started telling others about it, so I began putting releases on an FTP site and answering email about it. The early releases were never "announced"; they just spread by word of mouth. After some time, a UK ISP volunteered to run a web site and mailing list, and it has continued to grow from there. There has been a continuous stream of comments and suggestions, and there are far more facilities in current releases than I ever planned at the start.

Although I make a point of maintaining a comprehensive reference manual, one thing that has been lacking is introductory and tutorial material. I kept hoping that somebody else would write something, but in the end I was asked to write this book. I hope it will make life easier for those who find the reference manual difficult to work with.

Organization of the Book

After a short overview chapter, this book continues with a general introduction to Internet email, because this is a subject that does not seem to be well covered elsewhere. The rest of the book is devoted to explaining how Exim works, and how you can use its configuration to control what it does. Here is a detailed breakdown of the chapters:

Chapter 1, Introduction
This chapter is a short "executive" summary.

Chapter 2, How Internet Mail Works
This chapter is a general introduction to the way email is handled on Internet systems.

Chapter 3, Exim Overview
This chapter contains a general overview of the way Exim works, and introduces you to the way it is configured, in particular in regard to the way messages are delivered.

Chapter 4, Exim Operations Overview
This chapter continues with more overview material, mostly about topics other than the delivery of messages.

Chapter 5, Extending the Delivery Configuration
In this chapter, we return to the subject of message delivery, and show how the configuration can be extended to support additional functionality.

Chapter 6, Options Common to Directors and Routers
This is the first of a sequence of chapters that cover Exim's directors, routers, and transports and their options in detail.

Chapter 7, The Directors
This chapter covers the directors, which are the components of Exim that determine how local addresses are handled.

Chapter 8, The Routers
This chapter describes the routers, which are the components of Exim that determine how remote addresses are handled.

Chapter 9, The Transports
This chapter discusses the transports, which are the components of Exim that actually transport messages.

Chapter 10, Message Filtering
: This chapter describes the filtering language that is used both by users' filter files and the system filter.

Chapter 11, Shared Data and Exim Processes
: This chapter describes the various different kinds of Exim processes, and the data that they share.

Chapter 12, Delivery Errors and Retrying
: This chapter is concerned with temporary delivery errors, and how Exim handles them.

Chapter 13, Message Reception and Policy Controls
: Up to this point, the bulk of the book is concerned with delivering messages. This chapter describes the facilities that are available for controlling incoming messages.

Chapter 14, Rewriting Addresses
: This chapter covers the facilities for rewriting addresses in messages as they pass through Exim.

Chapter 15, Authentication, Encryption, and Other SMTP Processing
: This chapter covers a number of topics that are concerned with the transmission and reception of messages using SMTP.

Chapter 16, File and Database Lookups
: This is the first of three chapters that go into detail about the three main facilities that provide flexibility in Exim's configuration. They are all introduced in earlier chapters, but full details begin here.

Chapter 17, String Expansion
: This chapter gives all the details about Exim's string expansion mechanism.

Chapter 18, Domain, Host, and Address Lists
: This chapter provides all the details about the three kinds of lists that can appear in Exim configurations.

Chapter 19, Miscellany
: This chapter collects a number of items that do not fit naturally into the other chapters, but are too small to warrant individual chapters of their own.

Chapter 20, Command-Line Interface to Exim
: This chapter gives details of the options and arguments that are used to control what a call to Exim actually does.

Chapter 21, Administering Exim
: This chapter discusses a number of topics concerned with administration, and describes the utility programs that are available to help with this, including the Exim monitor, which is an application for displaying information about Exim's activities in an X window.

Chapter 22, Building and Installing Exim
: This chapter describes how to build and install Exim from the source distribution.

Appendix A, Summary of String Expansion
: This appendix is a summary of string expansion items.

Appendix B, Regular Expressions
: This appendix is a full reference description of the regular expressions that are supported by Exim.

Conventions Used in This Book

The following is a list of the typographical conventions used in this book:

Italic
: Used for file and directory names, program and command names, host and domain names, email addresses, mail headers, and new terms.

Bold
: Used for names of Exim directors, transports, and routers.

`Constant Width`
: Used in examples to show the contents of files or the output from commands, and in the text to mark Exim options or other strings that appear literally in configuration files.

`Constant Italic`
: Used to indicate variable options, keywords, or text that the user is to replace with an actual value.

`Constant Bold`
: Used in examples to show commands or other text that should be typed literally by the user.

Comments and Questions

We have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions for future editions, by writing to:

O'Reilly & Associates, Inc.
101 Morris Street
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, or any additional information. You can access this page at:

http://www.oreilly.com/catalog/exim

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, software, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com

Acknowledgments

I could not have produced Exim without the support and assistance of many people and organizations. There are too many to acknowledge individually, even if I had been organized enough to keep a full list, which, to my regret, I have not done. I hope that I have not made any major omissions in what follows.

For Exim itself, I must first acknowledge my colleagues in Computing Service at the University of Cambridge. The management allowed me to write Exim, and once it appeared, Computing Service has supported its use around the university and elsewhere.

Piete Brooks was brave enough to put the first version into service to handle mail for the Cambridge computer scientists. Piete also implemented the scheme for compiling on multiple operating systems. Piete suggested that an integral filter would be a good thing. Alan Barratt provided the initial code for relay checking. Nigel Metheringham persuaded his employers at that time, Planet Online Ltd., to provide support for an Exim web site and mailing list. Although he no longer works for them, he still manages the site and the mailing lists, and Planet (now called Energis Squared) still provides hardware and network resources. Nigel also provided code for interfacing to the Berkeley DB library, for supporting cdb files, and for delivering to mailboxes in maildir format. Yann Golanski provided the code for the numerical hash function. Steve Clarke did experiments to determine the most efficient way of finding the load average in Linux. Philip Blundell implemented the first support for IPv6 while he was a student at Cambridge. Jason Gunthorpe provided additional IPv6 code for Linux. Stuart Lynne provided the first code for LDAP support; subsequent modifications came from Michael Haardt, Brian Candler, and Barry Pederson. Steve Haslam provided some preliminary code for supporting TLS/SSL. Malcolm Beattie wrote the interface for calling an embedded Perl interpreter. Paul Kelly wrote the original code for calling MySQL, and Petr ??ENTITY-Ccaronech did the same for PostgreSQL. Jeff Goldberg pointed out that I was using the word "fail" in two different senses in the Exim documentation, and suggested "decline" for one of them. John Horne reads every edition of the reference manual, and picks up my typos and other mistakes. Over the five years since the first Exim release, many other people have sent suggestions for improvements or new features, and fixes for minor problems.

Finally, I must acknowledge my debt to Smail 3, written by Ron Karr, on which I based the first versions of Exim. Though Exim has now changed to become almost unrecognizable, its parentage is still visible.

While writing this book, I have continued to enjoy the support of my colleagues and the Exim community. My wife Judith was not only generally supportive, but also read an early draft as a professional copyeditor, and found many places where I was unclear or inconsistent. Ken Bailey made some useful comments about some of the early chapters. John Horne read an early draft and made suggestions that helped me to put the material into a more accessible order, and then read the book again in a late draft, thereby providing further useful feedback.

My editor at O'Reilly is Andy Oram, whose comments and guidance have had a great effect on the form and shape of the finished book. Andy has prevented me from becoming too obfuscated, and he also stopped me when I was writing too much British English.

1

Introduction

Exim is a *mail transfer agent* (MTA) that can be run as an alternative to Sendmail on Unix systems.* Exim is open-source software that is distributed under the GNU General Public License (GPL), and it runs on all the most popular flavors of Unix and many more besides. A number of Unix distributions now include Exim as their default MTA.

I wrote Exim for use on medium-sized servers with permanent Internet connections in a university environment, but it is now used in a wide variety of different situations, from single-user machines on dial-up connections to clusters of servers supporting millions of customers at some large ISP sites. The code is small (between 500 KB and 1.2 MB on most hardware, depending on the compiler and which optional modules are included), and its performance scales well.

The job of a mail transfer agent is to receive messages from different sources and to deliver them to their destinations, potentially in a number of different ways. Exim can accept messages from remote hosts using SMTP† over TCP/IP, and as well as from local processes. It handles local deliveries to mailbox files or to pipes attached to commands, as well as remote SMTP deliveries to other hosts. Exim consists of support for the new IPv6 protocol in its TCP/IP functions, as well as for the current IPv4 protocol. It does not directly support UUCP, though it can be interfaced to other software that does, provided that UUCP "bang path" addressing is not required, because Exim supports only Internet-style, domain-based addressing.

* The terms mail transfer agent and mail transport agent are basically synonymous, and are used interchangeably.

† If you are not familiar with SMTP or some of the other acronyms used here, don't be put off. The next chapter contains a description of how Internet mail works.

Exim's configuration is flexible and can be set up to deal with a wide variety of requirements, including virtual domains and the expansion of mailing lists. Once you have grasped the general principles of how Exim works, you will find that the runtime configuration is straightforward and simple to set up. The configuration consists of a single file that is divided into a number of sections, and entries in each section that are keyword/value pairs. Regular expressions, compatible with Perl 5, are available for use in a number of options.

The configuration file can reference data from other files, in linear and indexed formats, and from NIS, NIS+, LDAP, MySQL, and PostgreSQL databases. It can also make use of online lists such as the *Realtime Blackhole List* (RBL).[*] By this means, you can make much of Exim's operation table-driven if desired. For example, it is possible to do local delivery on a machine on which the users do not have accounts. The ultimate flexibility can be obtained (at a price) by running a Perl interpreter while processing certain option strings.

You can use a number of different facilities for checking and controlling incoming messages. For example, the maximum size of messages can be specified, SMTP calls from specific hosts and networks (optionally from specific identifiers) can be locked out, as can incoming SMTP messages from specific senders You can identify blocked hosts explicitly, or via RBL lists, and you can control which hosts are permitted to use the Exim host as a relay for onward transmission of mail. The SMTP AUTH mechanism can be used to authenticate client hosts for this purpose.

End users are not normally concerned with which MTA is delivering into their mailboxes, but when Exim is in use, its filtering facility, which extends the power of the traditional *.forward* file, can be made available to them. A filter file can test various characteristics of a message, including the contents of the headers and the start of the body, and then direct delivery to specified addresses, files, or pipes according to what it finds. The filtering feature can also be used by the system administrator to inspect each message before delivery.

Like many MTAs, Exim has adopted the Sendmail command interface so that it can be a straight replacement for */usr/sbin/sendmail* or */usr/lib/sendmail*. All the relevant Sendmail options are implemented. There are also some additional options that are compatible with Smail 3, and some further options that are specific to Exim.

* See *http://mail-abuse.org/rbl/*.

Messages on the queue can be controlled by the use of certain privileged command-line options. There is also an optional monitor program called *eximon*, which displays current information in an X window, and contains interfaces to the command-line options.

Exim is not designed for storing mail for dial-up hosts. When the volumes of such mail are large, it is better to get the messages "delivered" into files (that is, off Exim's queue) and subsequently passed on to the dial-up hosts by other means.

There are some things that Exim does not do: it does not support any form of delivery status notification,* and it has no built-in facilities for modifying the bodies of messages. In particular, it never translates message bodies from one form of encoding to another.

The aim of this book is to explain how Exim works, and to give background and tutorial information on the core facilities that the majority of administrators will need to know about. Some options that are required only in very special circumstances are not covered. In any case, a book can never keep up with developing software; if you want to know exactly what is available in any given release, you should consult the reference manual and other documentation that is included in the distribution for that release.

Exim is still being developed in the light of experience, changing requirements, and feedback from users. This book was originally written to correspond to Release 3.16, but while it was being revised, additional facilities, such as support for LMTP and SSL/TLS, were added to Exim for the 3.20 release. Some references to these important new features have therefore been included in the book, which now covers all the major features of the 3.2*x* releases. No further functional enhancements to Exim 3 are planned, though in due course a new major release (Exim 4) is expected.

The Exim reference manual and a FAQ are online at the Exim web site, at *http://www.exim.org* and its mirrors. Here you will also find the latest release of Exim, as a source distribution. In addition to the plain text version that is included in the distribution, the manual can be downloaded in HTML (for faster browser access), in PostScript or PDF (for printing), and in Texinfo format for the *info* command.

* See RFC 1891.

Some versions of GNU/Linux are now being distributed with binary versions of Exim included. For this reason, I've left the material on building Exim from source until the end of the book, and concentrated on the runtime aspects first. If you are working with a binary distribution, make sure you have a copy of the text version of the reference manual that comes with the source distribution. It provides full coverage of every configuration option, and can easily be searched.

The next chapter is a general discussion of the way email on the Internet works; Exim is hardly mentioned. This material has been included for the benefit of the many people who find themselves having to run a mail server without this essential background knowledge. You can skip to Chapter 3, *Exim Overview* if you already know about RFC 822 message format, SMTP, mail routing, and DNS usage.

2

How Internet Mail Works

The programs that users use to send and receive mail (often just called "mailers") are formally called *mail user agents* (MUAs). They are concerned with providing a convenient mail interface for users. They display incoming mail that is in users' mailboxes, assist the user in constructing messages for sending, and provide facilities for managing folders of saved messages. They are the "front end" of the mail system. Many different user agents can be installed, and can be simultaneously operational on a single computer, thereby providing a choice of different user interfaces. However, when an MUA sends a message, it does not take on the work of actually delivering it to the recipients. Instead, it sends it to a *mail transfer agent* (MTA), which may be running on the same host or on some local server.

Mail transfer agents do the job of transferring messages from one host to another, and, after they reach their destination hosts, of delivering them into user mailboxes or to processes that are managing user mailboxes. This job is complicated, and it would not be sensible for every MUA to contain all the necessary apparatus. The flow of data from a message's sender to its recipient is as shown in Figure 2-1. However, when an application program or script needs to send a mail message as part of some automatic activity, it normally calls the MTA directly without involving an MUA.

Only one MTA can be fully operational on a host at once, because only one program can be designated to receive incoming messages from other hosts. It has to be a privileged program in order to listen for incoming TCP/IP connections on the SMTP port and to be able to write to users' mailboxes. The choice of which MTA to run is made by the system administrator, whereas the choice of which MUA to run is made by the end user.

An MTA must be capable of handling many messages simultaneously. If it cannot deliver a message, it must send an error report back to the sender. An MTA must

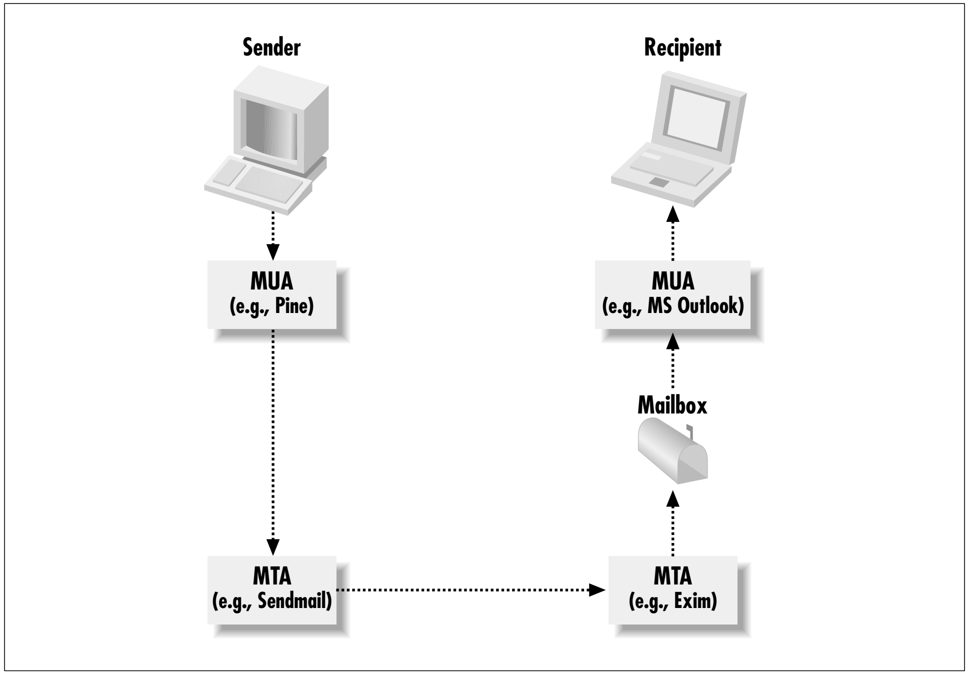


Figure 2-1. Message data flow

be able to cope with messages that cannot be immediately delivered, storing such messages on its local disk, and retrying periodically until it succeeds in delivering them or some configurable timeout expires. The most common causes of such delays are network connectivity problems and hosts that are down.

From an MTA's point of view, there are two sources of incoming messages: local processes and other hosts. There are three types of destinations: local files, local processes via pipes, and other hosts, as indicated in Figure 2-2.

The division of labor between MUAs and MTAs also means that an MUA need not be running on the same host as its MTA; Figure 2-3 illustrates the relationship between MUAs and MTAs in two common configurations.

In the top part of the figure, the MUA, MTA, and the disk storage are all part of a single system, indicated by the dashed line. The users access the system by logging on and authenticating themselves by a password or some other means. The MUA is started by a user command as a process on the system, and when it passes

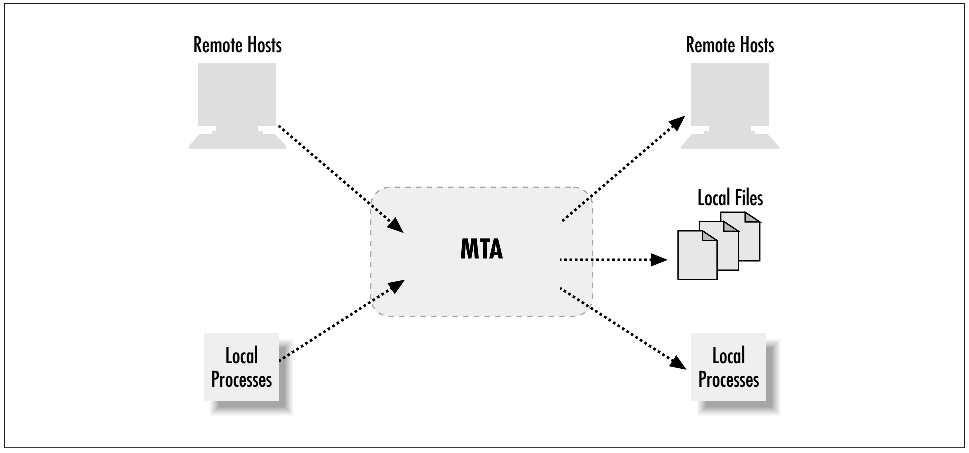


Figure 2-2. The job of an MTA

a message to the MTA for delivery, it is communicating with another process on the same system. Consequently, both the MUA and the MTA know the authenticated identity of the message's sender, and the MTA can ensure that this identity is included in the outgoing message. As specified in RFC 822,* if the contents of the *From:* header line do not match the actual sender, the MTA should normally add a *Sender:* line containing the authenticated identity.†

Messages are held by the MTA in its spool area while awaiting delivery. The word "spool" is often used with two different meanings. In this book, we use it to mean the disk storage that an MTA uses for messages that it has in transit. You will sometimes see "spool" used for the disk area in which users' mailboxes are kept, but this is not the sense in which it is used here.

Messages that are destined for other hosts are transmitted over the Internet to other MTAs using the *Simple Mail Transfer Protocol* (SMTP). When the originating host and the final host are both directly connected to the Internet, the message can be delivered directly to the final host, but sometimes it has to travel via an intermediate MTA. Large organizations often arrange for all their incoming mail to be routed via a central *mail hub*, which then delivers it to other hosts within the organization's local network. These may be behind a firewall and therefore inaccessible to the Internet at large. When a message reaches its destination host, the

* RFCs are the documents that lay down the standards by which the Internet operates. You can find them online at *http://www.ietf.org* (and numerous other places). We say a little bit about those that relate to mail later in this chapter.

† Exim does this by default, but can be configured not to.

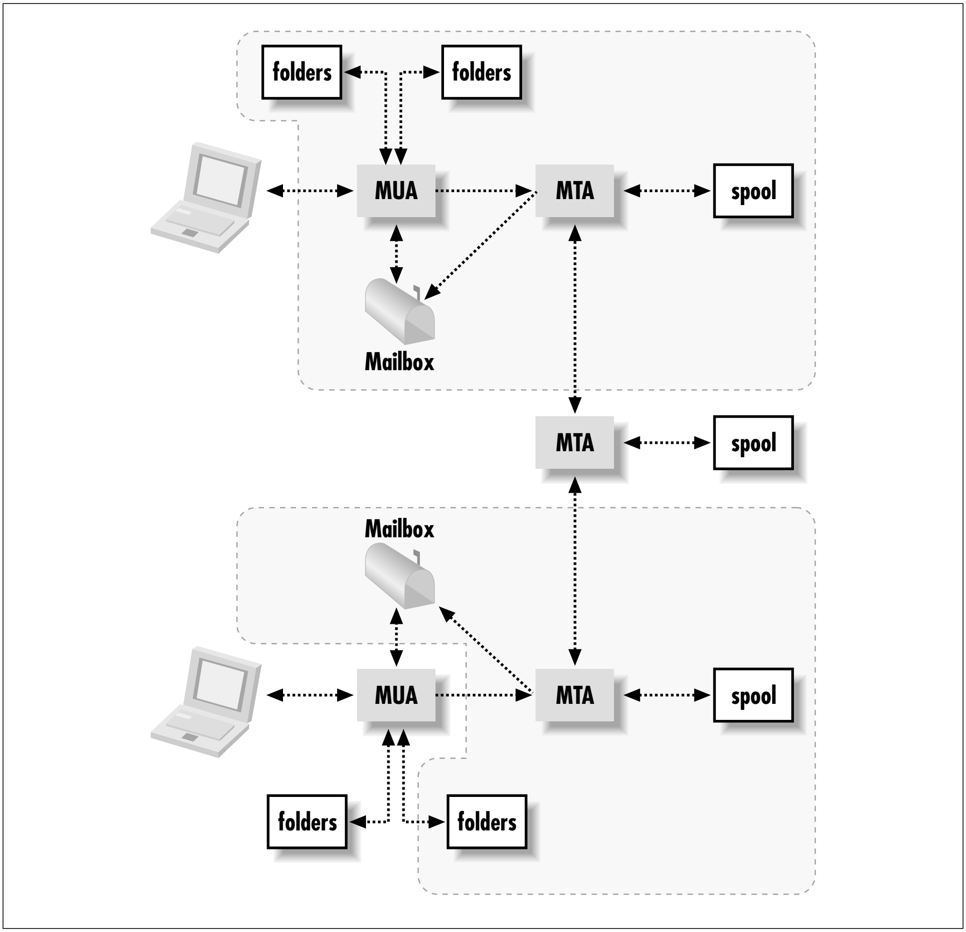


Figure 2-3. MUAs and MTAs

MTA delivers it into the mailbox of the recipient, who can then access it with the MUA of his choice.

Another case where an intermediate MTA is involved is when the final destination or its network connection is down. Using the *Domain Name Service* (DNS)* or some private method, a backup host may be designated for a domain. Incoming mail accumulates on this host until the main one starts working again, at which point the backlog is transferred. The advantage of this is that the accumulated mail can be stored close to the final destination, and can eventually be transferred quickly and in a controlled manner. In contrast, when a busy host without a backup restarts, it is liable to receive a very large number of simultaneous

* See RFCs 1034 and 1035.

incoming SMTP calls from all over the Internet, which may cause performance problems.

The bottom part of Figure 2-3 illustrates another popular configuration, in which the MUA is not running on the same system as the MTA. Instead it runs on a user's workstation. Receiving and sending messages in this configuration are entirely separate operations. When a user reads mail, the MUA uses either the POP (RFC 1939) or IMAP (RFC 2060) protocol to access the mailbox and remote folders on the server system. In order to do so, the user has to be authenticated in some way; commonly a username and password are used to gain access to the mailbox and remote mail folders. However, neither the POP nor IMAP protocols contain any facilities for sending messages. MUAs of this type have therefore traditionally used the SMTP protocol to pass messages to an MTA in a server system. Thus a protocol that was originally designed for passing messages between MTAs is subverted for the purpose of submitting new messages to an MTA, which is really a different kind of operation. This usage leads to a number of problems:

- The MTA cannot distinguish between a new message submission from an MUA and a message being passed on from another MTA. It may be able to make a guess, based on the IP address of local hosts it knows not to be running MTAs, but this is not always easy to arrange. This means that it cannot treat submissions specially, as it does when messages originate on the local host.
- The sender of the message is not authenticated; the MTA may be able to verify that the domain of the sender exists, but often it cannot check the local part of the address. MUAs of this type require the user to specify a username when starting; a typo made while doing this may go undetected, leading to incorrect sender addresses in outgoing messages.
- The MUA is not constrained to sending outgoing mail to the same server it is using for reading mail. It may sometimes be desirable to use different servers, but because of the existence of this flexibility, it is possible to direct MUA software to send mail to any host on the Internet. This makes it easy for unscrupulous persons to attempt to dump unsolicited mail on arbitrary servers for relaying. The fact that this has happened on numerous occasions has led to the tightening up of relaying servers, and the creation of databases such as the MAPS *Dialup User List*.*

* See *http://mail-abuse.org/dul/*.

There are some moves afoot to remedy this situation by defining a new submission protocol.* This is basically the same as SMTP, but it uses a different port number. However, at the time of writing, this technology is not yet in common use.

Different Types of MTA

The framework for mail delivery described earlier in this chapter is very general, and in practice there are many different kinds of MTA configuration that operate within it. At the simplest level, there are single hosts running in small offices or homes, each handling a few mailboxes in one domain, receiving incoming external messages from one ISP's mail server only, and sending all outgoing messages to the ISP for onward delivery. Many such hosts are not permanently connected to the Internet, but instead dial up from time to time to exchange mail with the server. In such an environment, the MTA does not have to be capable of doing full mail routing or complicated queue management.

Hosts that are permanently connected need not send everything via the same server, but can make use of the DNS to route outgoing messages more directly toward their final destinations. A single outgoing message may have several recipients, thus requiring copies to be sent to more than one remote server. This means that the MTA has to cope with messages where some of the addresses cannot be immediately delivered, and it must implement suitable retrying mechanisms for use with multiple servers. For incoming mail, the domain can be configured so that mail comes direct from anywhere on the Internet, without having to pass through an intermediate server.

An organization may not want to have all its local mailboxes on the same host. Even a small organization with just one domain may have users running their own desktop systems who want their mail delivered to them. The host running the "corporate" MTA has now become a *hub*, receiving mail from the world, and distributing it by user within its local network. It is common in such configurations for all outgoing mail from the network to pass through the hub. For security reasons, it is also common to configure the network router so that direct SMTP connections between the world and the workstations are not permitted.

Single organizations may support more than one domain, but the MTAs that support very large numbers of domains are usually those run by ISPs, and there are two common ways in which these are handled:

* See RFC 2476.

- For personal clients, the ISP normally provides a mailbox for each account, from which the mail is collected by some means when the client connects. As far as the MTA is concerned, it is doing a local delivery into a mailbox on the ISP's server.
- For corporate clients, ISPs are more likely to transfer mail to the clients' MTAs based purely on the domains in the addresses, with the ISP's MTA acting as a standard intermediate MTA between unrelated systems.

Internet Message Standards

Electronic mail messages on the Internet are formatted according to RFC 822, which defines the format of a message as it is transferred between hosts, but not the protocol that is used for the exchange. The Simple Mail Transfer Protocol (SMTP) is used to transfer messages between hosts. This is defined in RFC 821, with additional material in RFC 1123 and several other RFCs that describe extensions. The SMTP address syntax is more restrictive than that of RFC 822, and requires that components of domain names consist only of letters, digits, and hyphens. Since any message may need to be transported using SMTP if its destination is not on the originating host, the format of all addresses is normally restricted to what RFC 821 permits.

All these RFCs are now very old, and revised versions are nearing completion at the time of writing (February, 2001). The revisions consolidate the material from the earlier RFCs, and incorporate current Internet practice.*

RFC 822 Message Format

A message consists of lines of text, and when it is in transit between hosts, each line is terminated by the character *carriage return* (ASCII code 13) immediately followed by *linefeed* (ASCII code 10), a sequence that is commonly written as CRLF. Within a host, messages are normally stored for convenience in RFC 822 format. Many applications use the local operating system's convention for line termination when doing this, but some use CRLF. The normal Unix convention is to terminate lines with a single linefeed character, without a preceding carriage return.

* The new RFCs were released with the numbers 2821 and 2822 as this book went to press.

A message consists of a *header* and a *body*. The header contains a number of lines that are structured in specific ways as defined by RFC 822. The following examples are the header lines that are commonly shown to someone who is composing a message, and will be familiar to any email user:

```
From: Philip Hazel <ph10@exim.example>
To: My Readers <all@exim.book.example>,
    My Loyal Fans <fans@exim.example>
Cc: My Personal Assistant <cwbaft@exim.example>
Subject: How electronic mail works
```

An individual header line can be continued over several actual lines by starting the continuations with whitespace. The entire header section is terminated by a blank line. The body of the message then follows. In its simplest form, the body is unstructured text, but later RFCs (MIME, RFC 1521) define additional header lines that allow the body to be split up into several different parts. Each part can be in a different encoding, and there are standard ways of translating binary data into printable characters so that it can be transmitted using SMTP. This is the mechanism that is used for message "attachments."

RFC 822 permits many variations for addresses that appear in message header lines. For example:

```
To: caesar@rome.example.com
To: Julius Caesar <caesar@rome.example.com>
To: caesar@rome.example.com (Julius Caesar)
```

Text in parentheses anywhere in the line is a comment. This applies to all header lines whose structure is constrained by the RFC, not just those header lines that contain addresses. For example, in the following:

```
Date: Fri, 7 Jan 2000 14:20:24 -0500 (EST)
```

the time zone abbreviation is a comment as far as RFC 822 formatting is concerned. Along with the generally available parenthetical comments, headers that contain addresses may contain a sequence of words before an actual address in angle brackets; these are normally used for descriptive text such as the recipient's full name. When a header line contains more than one address, a comma must be used to terminate all but the last of them.*

The terms *local part* and *domain* are used to refer to the parts of a mail address that precede and follow the @ sign, respectively. In the address *caesar@rome.example.com*, the local part is *caesar* and the domain is *rome.example.com*. The local part is often a username, but because it can also be an

* Some MUAs allow lists of recipients to be given using spaces as separators, but when such a list is used to construct a *To:*, *Cc:*, or *Bcc:* header line, commas must be inserted.

abstraction such as the name of a mailing list or an address in some other mail domain in a message that is being sent to a gateway, the more general term is used here, as it is in the Exim reference documentation.

The Message "On the Wire"

A message that is transmitted between MTAs has several things added to it over and above what the composing user sees. In addition to the header section and the body, another piece of data called the *envelope* is transmitted immediately before the RFC 822 data, using the SMTP commands `MAIL` and `RCPT`. The envelope contains the sender address and one or more recipient addresses. These addresses are of the form <*user@domain*> without the additional textual information, such as the user's full name, that may appear in message header lines.

The deliveries done by the receiving MTA (either to local mailboxes or by passing the message on to other hosts) are based on the recipients listed in the envelope, not on the *To:* or *Cc:* header lines in the message. If any delivery fails, it is to the envelope sender address that the failure report is sent, not the address in the *From:* or *Reply-to:* header line.

The need for a separate envelope becomes obvious when considering a message with multiple recipients, whose mailboxes may be on several different hosts. The RFC 822 header lines normally list all the recipients, but in order to be delivered, the message has to be cloned into separate copies, one for each receiving host, and in each copy the envelope contains just those recipients whose mailboxes are on that host.

As well as an envelope, additional header lines are added by both the MUA and MTA before a message is transmitted to another host. Here is an example of a message "in transit," where the envelope lists only two of the three recipients. This example shows just the SMTP commands and data that the client sends, without the responses from the server:*

```
MAIL FROM:<ph10@exim.example>
RCPT TO:<fans@exim.example>
RCPT TO:<cwbaft@exim.example>
DATA
Received: from ph10 by draco.exim.example with local (Exim 3.22 #1)
        id 14Tli0-000501-00;
        Fri, 16 Feb 2001 14:18:05 +0000
From: Philip Hazel <ph10@exim.example>
To: My Readers <all@exim.book.example>,
    My Loyal Fans <fans@exim.example>
Cc: My Personal Assistant <cwbaft@exim.example>
```

* More details of the SMTP protocol are given in the next section of this chapter.

```
Subject: How electronic mail works
Date: Fri, 16 Feb 2001 14:18:05 +0000
Message-ID: <Pine.SOL.3.96.990117111343.19032A-100000@
  draco.exim.example>
MIME-Version: 1.0
Content-Type: TEXT/PLAIN; charset=US-ASCII

Hello,
  If you want to know about Internet mail, look at chapter 2.
.
```

The first three lines are the envelope; the message itself follows the DATA command, and is terminated by a line containing just a dot. Notice that lines have been added at both the start and the end of the header section.

Before passing a message to an MTA, an MUA normally adds *Date:* (required by RFC 822) and *Message-id:*. The MUA may also add header lines such as *MIME-Version:* and *Content-Type:* if the body of the message is structured according to the MIME definitions. Each MTA through which a message passes adds a *Received:* header line at the front, as required by RFC 821. The routing history of a message can therefore be obtained by reading these header lines in reverse order.

Because there may be quite a number of "behind-the-scenes" header lines by the time a message is delivered, most MUAs normally show only a subset when displaying a message to a user (typically the lines containing addresses, the subject, and the date). However, there is usually some way to configure the MUA to show all the header lines.

A recipient address that appears in the envelope need not appear in any header line in the message itself. This is usually the case after a message has passed through a mailing list expander, and is also the means by which "blind carbon copies" are implemented. When a user sends a message, either the MUA or the first MTA creates the envelope, taking the recipients from the *To:*, *Cc:*, and *Bcc:* data, and removing any *Bcc:* header line, unless there are no other recipients, in which case an empty *Bcc:* header line is retained.* An alternative permitted implementation is to retain the *Bcc:* header line only in those copies of the message that are transmitted to *Bcc:* recipients.

When a message is delivered into a user's mailbox, some MTAs, including Exim (as normally configured), add an *Envelope-to:* header line giving the envelope recipient address that was received with the message. This can be helpful if the final envelope recipient does not appear in the header lines. For example, consider a message sent from a mailing list to an address such as *postmaster@xyz.example*, which is handled by an alias. Messages from mailing lists do not

* RFC 822 does not permit empty *To:* or *Cc:* header lines; if there are no relevant addresses, these lines must be omitted. Only *Bcc:* may appear with no addresses.

normally contain the recipient in any of the header lines. Instead, there is likely to be a line such as:

```
To: some-list@listdomain.example
```

The address *postmaster@xyz.example* appears only in the envelope. Suppose that aliasing causes this message to be delivered into the mailbox of the user called *pat*, who is the local postmaster. Without the addition of *Envelope-to:*, there is nothing in the message itself that indicates why it ended up in Pat's mailbox.

The envelope sender is also known as the *return path*, because of its use for returning delivery failure reports. In most personal messages, it is identical to the address in the *From:* header, but it need not be. There are two common cases where it differs:

- When a message is sent to a mailing list, the original envelope sender that was received with the message is normally replaced with the address of the list manager before the message is sent out to the subscribers. This means that any delivery failures are reported to the list manager, who can take appropriate action, rather than to the original sender, who cannot.
- Delivery failure reports (often called "bounce messages") that are generated by MTAs are sent out with empty envelope sender addresses. These often appear in listings as <>. This convention is used to identify such messages as bounces, so that if they in turn fail to get delivered, no subsequent failure report is generated. The reason for this is to avoid the possibility of mail bounce loops occurring.

When a message is delivered into a user's mailbox, Exim (as normally configured) adds a *Return-path:* header, in which it records the envelope sender.

Summary of the SMTP Protocol

SMTP is a simple command-reply protocol. The client host sends a command to the server, and then waits for a reply before proceeding to the next command.[*] Replies always start with a three-digit decimal number; for example:

```
250 Message accepted
```

The text is usually information intended for human interpretation, though there are some exceptions, where the number encodes the type of response. The first digit is the most important, and is always one of those shown in Table 2-1.

[*] There is an optional optimization called "pipelining," which allows batches of commands to be sent, and batches of replies to be received, but this is purely to improve performance. The overall behavior remains the same, and we describe only the simple case here.

Table 2-1. SMTP Response Codes

Code	Meaning
2*xx*	The command was successful
3*xx*	Additional data is required for the command
4*xx*	The command suffered a temporary error
5*xx*	The command suffered a permanent error

The second and third digits give additional information about the response, but an MTA need not pay any attention to them. Exim, for example, operates entirely on the first digit of SMTP response codes. Replies may consist of several lines of text. For all but the last of them, the code is followed by a hyphen; in the last line it is followed by whitespace. For example:

```
550-Host is not on relay list
550 Relaying prohibited by administrator
```

When a client connects to a server's SMTP port (port 25), it must wait for an initial success response before proceeding. Some servers include the identity of the software they are running (and maybe other information) in the response, but none of this is actually required. Others send a minimal response such as:

```
220 ESMTP Ready
```

The client initializes the session by sending an `EHLO` (extended hello) command, which gives its own name.* For example:

```
EHLO client.example.com
```

Unfortunately, there are many MTAs in use that are misconfigured, either accidentally or deliberately, such that they do not give their correct name in the `EHLO` command. This means that the data obtained from this command is not of much use. The server's response to `EHLO` gives the server's name in the first line, optionally followed by other information text, and lists the extended SMTP features that the server supports in subsequent lines. For example, the following:

```
250-server.example.com Hello client.example.com
250 SIZE 10485760
```

indicates that the server supports the `SIZE` option, with a maximum message size of 10,485,760 characters.

Once an `EHLO` command has been accepted, the client may attempt to send any number of messages to the host. Each message is begun by a `MAIL` command,

* The original SMTP protocol used `HELO` (sic) as the initializing command, and servers are still obliged to recognize this. The difference is that the response to `EHLO` includes a list of the optional SMTP extensions that the server supports.

which contains the envelope sender address. If the `SIZE` option is supported by the server, the size of the message may also be given. For example:

```
MAIL FROM:<caesar@rome.example> SIZE=12345
```

After this has been accepted, each recipient address is transmitted in a separate `RCPT` command such as:

```
RCPT TO:<brutus@rome.example>
```

The client waits for a reponse to each one before sending the next. The server may accept some recipients and reject others, either permanently or temporarily. After a permanent error, the client should not attempt to resend the message to that address. The most common reasons for permanent rejection are as follows:

- The address contains a domain that is local to the server, but the local part is not recognized.
- The address contains a domain that is not local to the server, and the client is not authorized to relay through the server to that domain.

Temporary errors are caused by problems that are expected to be resolved in due course, such as the inability to check an incoming address because a database is down, or a lack of disk space. After a temporary error, a client is expected to try the address again in a new SMTP connection, after a suitable delay. This is normally at least 10 or 15 minutes after the first failure; if the temporary error condition persists, the time between retries is usually increased.

Provided that at least one recipient has been accepted, the client sends:

```
DATA
```

and the server responds with a 354 code, requesting further data; namely, the message itself. The client transmits the message without waiting for any further responses, and ends it with a line containing just a single dot character. If the message contains any lines that begin with a dot, an extra dot is inserted to guard against premature termination. The server strips a leading dot from any lines that contain more text. If the server returns a success response after the data has been sent, it assumes responsibility for subsequent handling of the message, and the client may discard its copy of it. Once it has sent all its messages, a client ends the SMTP session by sending a `QUIT` command.

Because SMTP transmits the envelope separately from the message itself, servers can reject envelope addresses individually, before much data has been sent. However, if a server is unhappy with the contents of a message* it cannot send a rejection until the entire message has been received. Unfortunately, some client

* For example, if the message is too big, or if the server is configured to check the syntax of addresses in header lines and comes across one containing invalid syntax.

software (in violation of RFC 821) treats any error response to DATA or following the data itself as a temporary error, and continues to try to deliver the message at successive intervals.

Forgery

It is trivial to forge unencrypted mail. In general, MTAs are "strangers" to each other, so there is no way a receiving MTA can authenticate the contents of the envelope or the message itself. All it can do is log the IP address of the sending host, and include it in the *Received:* line that it adds to the message.

Unsolicited junk mail (spam) usually contains some forged header lines. You need to be aware of this if you ever have to investigate the origin of such mail. If a message contains a header line such as:

```
Received: from foobar.com.example ([10.9.8.7])
        by podunk.edu.example (8.9.1/8.9.1) with SMTP id DAA00447;
        Tue, 6 Mar 2001 03:21:43 -0500 (EST)
```

it does not mean that the FooBar company or the University of Podunk are necessarily involved at all; the header may simply have been inserted by the spam perpetrator to mislead. The only *Received:* headers you can count on are those at the top of the message that were added by MTAs running on hosts whose administrators you trust. Once you pass these *Received:* headers, those below them, even if they appear to relate to a reputable organization such as an ISP, may be forged.

Authentication and Encryption

The original SMTP protocol had no facilities for authenticating clients, nor for encrypting messages as they were transmitted between hosts. As the Internet expanded, it became clear that these features were needed, and the protocol has been extended to allow for them. However, the vast majority of Internet mail is still transmitted between unauthenticated hosts, over unencrypted connections. For this reason, we won't go into any details in this introductory chapter, but there is some discussion in Chapter 15, *Authentication, Encryption, and Other SMTP Processing*, regarding the way Exim handles these features.

Routing a Message

The most fundamental part of any MTA is the apparatus for deciding where to send a message. There may be many recipients, both local and remote. This means that a number of different copies may need to be made and sent to different destinations. Some domains may be known to the local host and processed specially; the remainder normally causes copies of the message to be sent to remote hosts, which may either be the final destinations or intermediate hosts.

There are two distinct types of address: those for which the local part is used when deciding how to deliver the message, and those for which only the domain is relevant. Typically, when a domain refers to a remote host, the local part of the address plays no part in the routing process, but if the domain is the name of the local host, the local part is all-important. The steps that an MTA has to perform in order to handle a message are as follows, though they are not necessarily done in this order:

- First, it has to decide what deliveries to do for each recipient address. In order to do this, it must:
  - Process addresses that contain domains for which this host is the ultimate destination. These are often called "local addresses." Processing may involve expanding aliases into lists of replacement addresses, handling users' *.forward* files, dealing with mailing lists, and checking that the remaining local parts refer to existing local user mailboxes.
  - Process the nonlocal addresses for which there is local routing knowledge (for example, domains for which the host is a mail hub or firewall) to determine which of its clients' hosts these addresses should be sent to.
  - For the remaining addresses, those for which there is no local knowledge, look up destination hosts in the DNS. The details of how this is done are given in the section, "DNS Records Used for Mail Routing," later in this chapter. Successful routing produces a list of one or more remote hosts for each address.
- After sorting out what deliveries need to be done, the MTA must carry out the local deliveries; that is, deliveries to pipes or files on the local host.
- Then, for each remote delivery, it must try to send to each host in turn, until one succeeds or gives a hard failure. If several addresses are routed to the same set of hosts, the RFCs recommend sending a single copy with multiple recipients in the envelope.
- If all hosts give temporary failures, the MTA must try the corresponding addresses again later. There is a timeout, normally a few days, to stop a client from retrying forever.

Checking Incoming Mail

Some MTAs check the validity of local addresses during the SMTP transaction. If an incoming message has an incorrect local part, the `RCPT` command that transfers that part of the envelope is rejected by giving an error reponse. This means that the sending MTA retains control of the message for that recipient, and is the one that generates the bounce message that goes back to the sender. The benefit of doing this checking is that it stops such undeliverable messages from ever getting into the local host. However, receiving a bounce message from an MTA that is not at the site they were mailing to confuses some users, and makes them think that something is broken. "How can the local mailer daemon know that this is an invalid address at the remote site?" they ask.

The alternative approach that is adopted by some MTAs is to accept messages without checking the recipient addresses, and do the checking later. This has the benefit of minimizing the duration of the SMTP transaction, and for invalid addresses, the bounce messages are what the users intuitively expect, and they can be made to contain helpful information about finding correct mail addresses. The disadvantage is that undeliverable messages whose envelope senders are also invalid give rise to undeliverable bounce messages that have to be sorted out by the postmaster. Sadly, many spam messages are sent out with invalid envelope senders, leading to more and more administrators configuring their MTAs to implement the former behavior.

Exim can be configured to behave in either of these two ways, and the behavior can be made conditional on the domain of the sender address. For example, all addresses from within a local environment can be accepted, and unknown ones passed to a program that sends back a helpful message, while unknown addresses from the outside can be rejected in the SMTP protocol.

Not all MTAs check the validity of envelope sender addresses. These can be invalid for a number of reasons, such as:

- Misconfigured MUAs or MTAs. For an MUA running on a workstation, the user has to supply the sender address, while an MTA's configuration contains a default domain that it adds to local usernames to create sender addresses. In either case, a typo can render the address invalid. Errors can also arise in gateways that are converting messages from some other protocol regime. Neverthless, such messages sometimes have a valid address in the *From:* header line.
- Use of domains not registered in the DNS.
- Misconfigured DNS name servers; for example, a typo in a zone file.
- Forgery.

In general, the checking of sender addresses is normally confined to verifying that the domain is registered in the DNS. It is not normally practicable to verify the local parts of remote addresses.*

* Exim 3.20 does contain a facility for making a "callback" to verify that an incoming sender address is acceptable as a recipient to a host that handles its domain, but this is a costly approach that is not suitable for use on busy systems.

The enormous increase in the amount of unsolicited mail being transmitted over the Internet has caused MTA implementors to add facilities for blocking certain types of message as a matter of policy. Typical features include the following:

- Checking local lists of known miscreant hosts and sender addresses.
- Checking one or more of the public "blacklists," such as the Realtime Blackhole List (*http://mail-abuse.org*), and either refusing messages from blacklisted hosts, or annotating them by adding an informational header line.
- Blocking third-party relaying through the local host. That is, preventing arbitrary hosts from sending mail to the local host for onward transmission to some other destination. MTAs that do not block such mail are called "open relays" and are a favorite target of spammers.
- Refusing messages with malformed header lines.
- Recognizing junk mail by scanning the content, and either discarding it or annotating it to inform the recipient, who then has the choice of discarding it by means of a filter file.
- Checking for certain types of attachments in order to block viruses.

Overview of the DNS

The DNS is a worldwide, distributed database that holds various kinds of data indexed by keys that are called *domain names*. Here is a very brief summary of the facilities that are relevant to mail handling.* The data is held in units called *records*, each containing a number of items, of which the following are relevant to applications that use the DNS:†

```
<domain name>  <record type>  <type-specific data>
```

For example, for the record:

```
www.web.example.  A  10.8.6.4
```

the domain name is *www.web.example*, the record type is "A" (for "address"), and the data is 10.8.6.4. Address records like this are used for finding the IP addresses of hosts from their names, and are probably the most common type of DNS record.

* For a full discussion, see *DNS and BIND* by Paul Albitz and Cricket Liu (O'Reilly). The primary DNS RFCs are 1034 and 1035.

† The other fields are concerned with the internal management of the DNS itself.

In the world of the DNS, a complete, fully qualified domain name is always shown with a terminating dot, as in the previous example. Incomplete domain names, without the trailing dot, are relative to some superior domain. Unfortunately, there is confusion because some applications that interact with the DNS do not show or require the trailing dot. In particular, domains in email addresses must *not* include it, because that is contrary to RFC 821/822 syntax.

The present Internet addressing scheme, which uses 32-bit addresses and is known as IPv4, is going to be replaced by a new scheme called IPv6, which uses 128-bit addresses. Support for IPv6 is gradually beginning to appear in operating systems and application software. Two different DNS record types are currently used for recording IPv6 addresses, which are normally written in hexadecimal, using colon separators. The AAAA record, which is a direct analogue to the A record, was defined first. For example:

```
ipv6.example.  AAAA  5f03:1200:836f:0a00:000a:0800:200a:c031
```

However, it has been realized that a more flexible scheme, in which prefix portions of IPv6 addresses can be held separately, is preferable, because it makes aggregation and renumbering easier. For this reason, another record type, A6, has been defined and is expected in due course to supersede the AAAA type. The previous example could be converted into a single A6 record such as this:

```
ipv6.example.  A6  0  5f03:1200:836f:0a00:000a:0800:200a:c031
```

The zero value indicates that no additional prefix is required. Alternatively, the address could have its prefix recorded in a separate record, like this:

```
ipv6.example.  A6  64  ::000a:0800:200a:c031 pref.example.
pref.example.  A6   0  5f03:1200:836f:0a00::
```

The value of 64 indicates that an additional 64 bits of prefix are required, and domain name *pref.example* identifies another A6 record where this prefix can be found. Several levels of prefix are permitted.*

If a host has more than one IP interface, each appears in a separate address record with the same domain name. The case of letters in DNS domain names is not significant, and the individual components of a name may contain a wide range of characters. For example, a record in the DNS could have the domain name *abc_xyz#2.example.com*. However, the characters that are used for hostnames are restricted by RFC 952 to letters, digits, and hyphens, and domains that are used in email addresses in the SMTP protocol are similarly restricted. It is not possible to

* See RFC 2874 for further details of how A6 records work.

send mail to an address such as *user@abc_xyz#2.example.com* using SMTP, because of the characters in the domain that are illegal according to RFC 821. For this reason, all MX domain names (which are described shortly) and hostnames use only the restricted character set. This constraint is often misunderstood to be an internal DNS restriction, which it is not.*

The servers that implement the DNS are called *name servers*, and are distributed throughout the Internet. The hierarchical name space is broken up into *zones*, each of which is managed by its own human administrator, and stored on its own master server. Division into zones that are stored on independent servers is what makes the management of such a large set of data practicable.†

The breakpoints between zones are always between components of a domain name, but not necessarily at every boundary. For example, there is a *uk* zone, and *ac.uk* and *cam.ac.uk* zones, but there is no separate *csx.cam.ac.uk* zone, although there are domain names ending with those components. The data for those names is held within the *cam.ac.uk* zone. This does not prevent there being other different zones below *cam.ac.uk*. This example is illustrated in Figure 2-4, using dashed lines to represent the zones.

There is usually one master name server for a zone, and several slaves that copy their data from the master. A single name server may be a master for some zones and a slave for others. Any name server (master or slave) that has its own complete copy of a zone file is said to be *authoritative* for that zone. You will sometimes see this word used in output from commands such as `host`, which interrogate the DNS. Data that a name server has obtained from some other name server without transferring the entire zone is nonauthoritative.

It is preferable for the slaves to be at least on different LANs to the master, and best if some of them are at entirely different locations in order to maximize the availability of the zone for queries. ISPs commonly provide name server slaving facilities for their customers. A name server for a zone that has subzones knows the location of the servers for those zones. At the base of the hierarchy are the *root* name servers at "well-known" locations on the Internet.

Caching is extensively used in DNS software to improve performance. Each record contains a time-to-live field, and name servers are entitled to remember and reuse the data for that length of time. A typical time-to-live is around one day. Data is looked up by passing the domain name and type to a nearby name server; if there has been a recent request for the same data, it will be in the server's cache and the request can be answered immediately. Otherwise, if the server happens to have

* See also RFC 2181.

† Before the DNS, the list of Internet hosts was kept in a single file that had to be copied in its entirety to all of them.

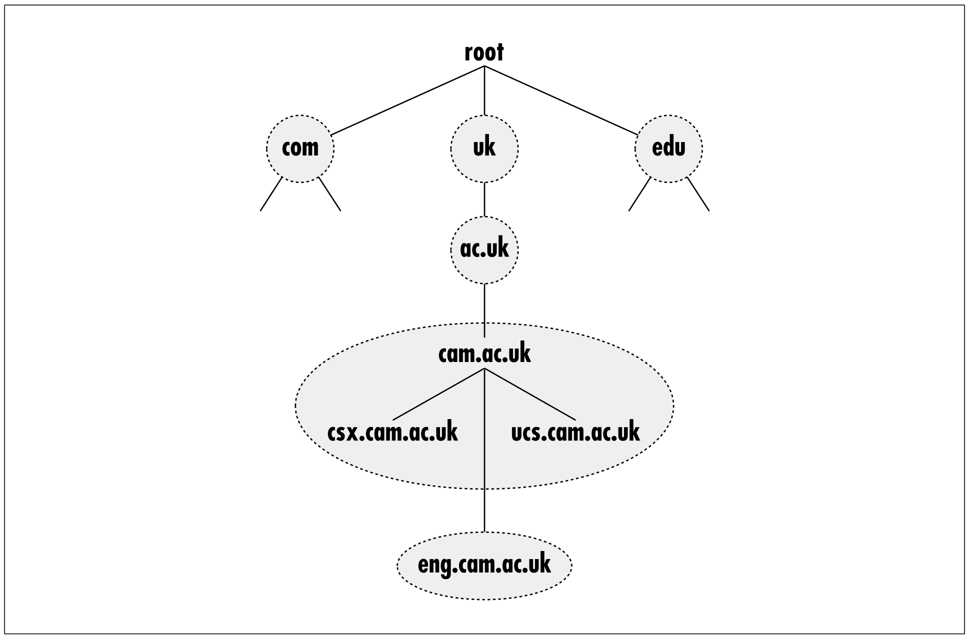


Figure 2-4. DNS domains and zones

cached the identity of the name servers for the required zone, it can query them directly, but if it has no relevant information, it starts by querying one of the root name servers and works its way down the zone hierarchy. For example: if a query for *www.cam.ac.uk* is received by a root name server, it responds with the list of name servers for the *uk* zone. Querying one of them produces a list of name servers for the *ac.uk* zone, and so on, until a name server that contains the actual data is reached.

DNS Records Used for Mail Routing

The domain in a mail address need not correspond to a hostname. For example, an organization might use the domain *plc.example.com* for all its email, but handle it with hosts called *mail-1.plc.example.com* and *mail-2.plc.example.com.* This kind of flexibility is obtained by making use of *mail exchange* (MX) records in the DNS. An MX record maps a mail domain to a host that is registered as handling mail for that domain, with a preference value. There may be any number of MX records for a domain, and when a name server is queried, it returns all of them. For example:

```
hermes.example.com.  MX  5  green.csi.example.com.
hermes.example.com.  MX  7  sw3.example.com.
hermes.example.com.  MX  7  sw4.example.com.
```

shows three hosts that handle mail for *hermes.example.com.* The preference values can be thought of as distances from the target; the smaller the value, the more preferable the corresponding host, so in this example, *green.csi.example.com* is the most preferred. An MTA that is deliverying mail for *hermes.example.com* first tries to deliver to *green.csi.example.com*; if that fails, it tries the less preferred hosts in order of their preference values. It is only the numerical order of the preferences that is used; the absolute values do not matter. When there are MX records with identical preference values (as in the previous example), they are ordered randomly before they are used.

Before an MTA can make use of the list of hosts it has obtained from MX records, it first has to find the IP addresses for the hosts. It does this by looking up the corresponding address records (A records for IPv4, and AAAA or A6 records for IPv6). For the previous example, there might be the following address records:

```
green.csi.example.com.  A  192.168.8.57
sw3.example.com.        A  192.168.8.38
sw4.example.com.        A  192.168.8.44
```

In practice, if a name server already has an address record for any host in an MX list that it is returning, it sends the address record along with the MX records. In many cases, this saves an additional DNS query.

In the early days of the DNS there were no MX records, and mail domains corresponded to hostnames. For backward compatibility to that time, if there are no MX records for a domain, an MTA is entitled to look for an address record and treat it as if it were obtained from an MX record with a preference value of zero (most preferred). However, if it cannot determine whether or not there are any MX records (because, for example, the relevant name servers are unreachable), it must not do this.

MX records were originally invented for use by gateways to other mail systems, but nowadays they are heavily used to implement "corporate" mail domains that do not necessarily correspond to any specific host.

Related DNS Records

Two other kinds of DNS records are useful in connection with mail. PTR ("pointer") records map IP addresses to names via special zones called *in-addr.arpa* for IPv4 addresses, and *ip6.int* or *ip6.arpa* for IPv6 addresses.* PTR records allow the reverse of a normal host lookup: given an IP address, PTR records allow you to find out the corresponding hostname. The name of a PTR record consists of the IP address followed by one of the special domains. However, for the *in-addr.arpa* and *ip6.int* domains, the components of the address are

* The top-level domain name *arpa* is rooted in history, and refers to the original wide-area network called the Arpanet.

reversed to allow for DNS delegation of parts of an IP network. For the address 192.168.8.57, the PTR record would be as follows:

```
57.8.168.192.in-addr.arpa.  PTR  green.csi.example.com.
```

This registers that the name of the host that has the IP address 192.168.8.57 is *green.csi.example.com.* For IPv6 addresses in the *ip6.int* domain, the components that are reversed are the hexadecimal digits. For the address:

```
5f03:1200:836f:0a00:000a:0800:200a:c031
```

the name of the PTR record is:

```
1.3.0.c.a.0.0.2.0.0.8.0.a.0.0.0.0.0.a.0.f.6.3.8.0.0.2.1.3.0.f.5.ip6.int.
```

Not only is this rather clumsy to notate, it also has the disadvantage that DNS zone breaks are not possible at arbitrary points in the 128-bit address. For this reason, at the time A6 records were introduced for name-to-address lookups, an alternative format for IPv6 PTR records was defined for use with the domain *ip6.arpa.* In this form, part of the domain name is a binary number with an implied component break between each binary digit. For convenience, in textual versions of the record, the number is given in conventional notation without having to be reversed. In this new formulation, the name of the PTR record in the previous IPv6 address is:

```
\[x5f031200836f0a00000a0800200ac031].ip6.arpa.
```

where the backslash and brackets indicate an encoding of a binary value.

PTR records do not have to match the corresponding address record. In the example in the previous section, the address record:

```
sw4.example.com.  A  192.168.8.44
```

is shown. If you use the address 192.168.8.44 to look up the hostname via a PTR record, you might find the name *sw4.example.com*, or you might find something completely different; for example:

```
44.8.168.192.in-addr.arpa.  PTR  lilac.csi.example.com.
```

This record gives the name *lilac.csi.example.com* for the address 192.168.8.44, despite the fact that the address was given for the name *sw4.example.com.* This kind of arrangement is often found where the name of some kind of service is widely published, with an address record to point to a host that is currently providing the service. The host itself, however, has a different primary name, which is what the PTR record contains.

For example, the name we've been using, *sw4.example.com*, might be the name of a mail switching service that currently is provided by the host *lilac.csi.example.com.* Moving the service to another host just requires the DNS to be updated; no host has to change its name. If more than one host is providing the service, several address records may exist for the same domain. Modern name servers return these in a different order each time they are queried, which provides a form of load-sharing.

There is no enforced connection between address records and PTR records, and for any given host, one may exist without the other. The main use of these records in connection with mail is for finding the name of the remote host that is sending a message, because all that is initially known about the host at the far end of an incoming TCP/IP call is its IP address. The hostname may be required for checking against policy rules controlling what types of message remote hosts may send.

CNAME ("canonical name") records provide another kind of aliasing facility. For example:

```
pelican.example.com.  CNAME  redshank.csx.example.com.
```

states that the canonical name (real or main name) for the host that can be accessed as *pelican.example.com* is actually *redshank.csx.example.com.* CNAME records should not normally be used in connection with mail routing. MX records provide sufficient redirection capabilities, and excessive aliasing just slows things down.

Common DNS Errors

These are a number of common mistakes that are made by DNS administrators (who are usually known as "hostmasters"), shown in the following list. All except the first prevent mail from being delivered:

- MX records point to aliases instead of canonical names. That is, the domains on the righthand side of MX records are the names of CNAME records instead of A, A6, or AAAA records. This should not prevent mail from working, but it is inefficient, and not strictly correct.
- MX records point to nonexistent hosts; that is, to names that have no corresponding A, A6, or AAAA record.
- MX records contain IP addresses on the righthand side instead of hostnames. This error is unfortunately becoming more widespread, abetted by the fact that some MTAs, in violation of RFC 1034, support the usage. Exim does not do so by default, but does have an option to enable this unrecommended, nonstandard behavior.
- MX records do not contain preference values.

Some broken name servers give a server error when asked for a nonexistent MX record. This prevents mail from being delivered because an MTA is permitted to search for an address record only if it is sure there are no MX records. In the case of a server error, the MTA does not know this. Similar server errors have been seen in cases where a preference value has been omitted from an MX record. More robust name servers check records when loading their zones, and generate an error if any contain bad data such as this.

Occasionally, the DNS appears to be giving different answers to identical queries. In the context of mail, this causes some messages to be rejected with "unknown domain" errors, whereas other messages to the same domain are delivered normally. The most common cause of this kind of behavior is that the name servers for the zone are out of step. If you suspect this, you can check by directing a DNS query to a specific name server. The first step is to find the relevant name servers by looking for the zone's NS records. To find the name servers for the zone *ioe.example.com*, for example, you can use the command:

```
$ nslookup -type=ns ioe.example.com
```

which might give these lines as the relevant parts of its answer:*

```
ioe.example.com    nameserver = mentor.ioe.example.com
ioe.example.com    nameserver = ns0.example.net
```

Once you know the name servers, you can query each one in turn for the domain in question; if the `nslookup` command is given a second argument, it is the name of a specific name server to which the query is to be sent. This sequence of commands and responses (where the commands are shown in boldface) indicates that there is a problem because the different name servers are giving conflicting answers:

```
$ nslookup saturn.example.com mentor.ioe.example.com
Server:  mentor.ioe.example.com
Address:  192.168.34.22

Name:    saturn.example.com
Addresses:  192.168.5.4
$ nslookup saturn.example.com ns0.example.net
Server:  ns0.example.net
Address:  192.168.255.249

*** ns0.example.net can't find saturn.example.com: Nonexistent host/domain
```

The problem may, however, be temporary. When a master name server is updated, it can take some hours before the data reaches the slaves, during which time this behavior may be seen. However, if the discrepancy persists for any length of time, it is indicative of some kind of DNS error.

* *nslookup* is one of the applications that omits the trailing dots when it displays domain names.

Role of the Postmaster

Postmaster is the name given to the person who is in charge of administering an MTA. He or she should be familiar with the software and its configuration, and should regularly monitor its behavior. If there are local users of the system, they should be able to contact the postmaster about any mail problems. If the MTA sends or receives mail to or from the Internet at large, people on other hosts must also be able to contact the postmaster.

The traditional way that this is done is by maintaining an alias address *postmaster@your.domain*, which redirects to the person who is currently performing the postmaster role. Indeed, the RFCs state that *postmaster* must always be supported as a case-insensitive local name.

3

Exim Overview

In the previous chapter, the job of an MTA is described in general terms. In this chapter, we explain how Exim is organized to do this job, and the overall way in which it operates. Then in the next chapter, we cover the basics of Exim administration before launching into more details about the configuration.

Exim Philosophy

Exim is designed for use on a network where most messages can be delivered at the first attempt. This is true for most of the time over a large part of the Internet. Measurements taken in the author's environment (a British university) indicate that well over 90 percent of messages are delivered almost immediately under normal conditions. This means that there is no need for an elaborate centralized queuing mechanism through which all messages pass. When a message arrives, an immediate delivery attempt is likely to be successful; only for a small number of messages is it necessary to implement a holding and retrying mechanism.

Therefore, although it is possible to configure Exim otherwise, the normal action is to try an immediate delivery as soon as a message has been received. In many cases this is successful, and nothing more is needed to process the message. Nevertheless, some precautions must be taken to avoid system overload in times of stress. For example, if the system load rises above some threshold, or if there are a large number of simultaneous incoming SMTP connections, immediate delivery may be temporarily disabled. In these events, incoming messages wait on Exim's queue and are delivered later.

All operations are performed by a single Exim binary, which operates in different ways, depending on the arguments with which it is called. Although receiving and delivering messages are treated as entirely separate operations, the code for determining how to deliver to a specific address is needed in both cases, because during message reception, addresses are verified by checking whether it would be possible to deliver to them. For example, Exim verifies a remote sender address by looking up the domain in the DNS in exactly the same way as when setting up a delivery to that address.

Exim's Queue

The word *queue* is used for the set of messages that Exim has under its control at any one time, because this word is common in the context of mail transfer. However, Exim's queue is normally treated as a collection of messages with no implied ordering, more like a "pool" than a "queue." Furthermore, Exim does not maintain separate queues for different domains or different remote hosts.

There is just a single collection of messages awaiting delivery, each of which may have several recipients. You can list the messages on the queue by running the command:

```
exim -bp
```

assuming that your path is set up to contain the directory where the Exim binary is located. Messages that are not delivered immediately on arrival are picked up later by *queue runner* processes that scan the entire queue and start a delivery process for each message in turn. A queue runner process waits for each delivery process to complete before starting the next one.

Receiving and Delivering Messages

Message reception and message delivery are two entirely separate operations in Exim, and their only connection is that Exim normally tries to deliver a message as soon as it has received it. Receiving a message consists of writing it to local spool files ("putting it on the queue") and checking that the files have been successfully written before acknowledging reception to the sending host or local process. There is only one copy of each message, however many recipients it has, and the collection of spool files *is* the queue; there are no additional files or in-memory lists of messages.

A delivery operation gets all its data from the spool files. Each attempt at delivering a message processes every undelivered recipient address afresh. Exim does not normally retain previous alias, forwarding, or mailing list expansions from one delivery attempt to another.*

* There is, however, one exception to this: if the `one_time` option is set for a mailing list, the list's addresses are added to the original list of recipients at the first delivery attempt, and no re-expansion occurs at subsequent attempts.

Exim Processes

Parallelism is obtained by the use of multiple processes, but one important aspect of Exim's design is that there is no central process that has overall responsibility for coordinating Exim's actions, and therefore there is no concept of starting or stopping Exim as a whole. Exim processes can be started at any time by other processes; for example, user agents are always able to start Exim processes in order to send messages. Such processes perform a single task and then exit. Most processes are therefore short-lived, but Exim does make use of long-running daemon processes for two purposes:

1. To listen on the SMTP port for incoming TCP/IP connections. On receiving such a connection, the listener forks a new process to deal with it. An upper limit to the number of simultaneously active reception processes can be set. When the limit is reached, additional SMTP connections are refused.
2. To start up queue runner processes at fixed intervals. These scan the pool of waiting messages (by default in an arbitrary order) and initiate fresh delivery attempts. A message may be on the queue because a previous delivery attempt failed, or because no delivery attempt was initiated when the message was received. Each delivery attempt processes a single message and runs in its own process, and the queue runner waits for it to complete before moving on to the next message. A limit may be set for the number of simultaneously active queue runner processes run by a daemon.

A single daemon process can be used to perform both these functions, and this is the most common configuration. However, it is possible to run Exim without using a daemon at all; *inetd* can be used to accept incoming SMTP calls and start up an Exim process for each one, and queue runner processes can be started by *cron* or some other means. However, in these cases Exim has no control over how many such processes are run, so if you are worried about system overload, you must control the number of processes yourself.*

Coordination Between Processes

Processes for receiving and delivering messages are for the most part entirely independent. The small amount of coordination that is needed is achieved by sharing files. Minimizing synchronization and serialization requirements between processes helps Exim to scale well. Apart from the messages themselves, the shared data

* *xinetd* (*www.xinetd.org*) is a replacement for *inetd* that includes additional control facilities.

consists of a number of files containing "hints" about mail delivery. For example, if a remote host cannot be contacted, the time of the failure and the suggested next time to try that host are recorded. Any delivery process that has a message for that host will read the hint and refrain from trying the delivery if the retry time has not been reached. This does not affect delivery of the same message to other hosts when there is more than one recipient address.

Because the coordinating data is treated as a collection of hints, it is not a major disaster if any or all of it is lost; there may be a period of less optimal mail delivery, but that is all. Consequently, the code that maintains the hints can be quite simple because it does not have to be made robust against unusual circumstances.

How Exim Is Configured

Configuration information, supplied by the administrator, is used at two different times: one configuration file is used when building the Exim binary, and another is read whenever the binary is run. Most options can be specified in only one of these files; that is, they either control how the binary is built, or they modify its behavior at runtime, but there are a few build-time options that set defaults for runtime behavior. The sources of Exim's configuration information are shown in Figure 3-1.

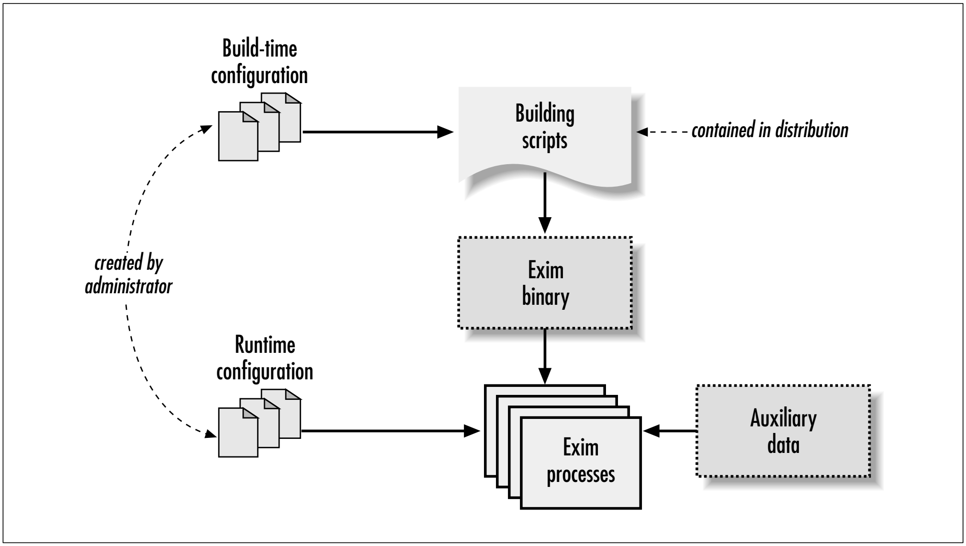


Figure 3-1. Exim configuration

The build-time options are of three kinds:

- Those that specify the inclusion of optional code; for example, to support specific database lookups such as LDAP, or to support IPv6.
- Those that specify fixed values that cannot be changed at runtime; for example, the mode of message files in Exim's spool directory.
- Those that specify default values for certain runtime options; for example, the location of Exim's log files.

The process of building Exim from source is described in detail in Chapter 22, *Building and Installing Exim*. Here, we consider the runtime configuration. This is controlled by a single text file, often called something like */etc/exim.conf*. You can find out the actual name by running the following command:

```
    exim -bP configure_file
```

On a system where Exim is fully installed as a replacement for Sendmail, one or both of the paths */usr/lib/sendmail* or */usr/sbin/sendmail* is a symbolic link to the Exim binary. Therefore, any MUA, program, or script that attempts to send a message by calling Sendmail actually calls Exim.*

Whenever Exim is executed, it starts by reading its runtime configuration file. A large number of settings can be present, but for any one installation only a few are normally used. The data from the file is held in main memory while an Exim process is running. For this reason, if you change the file, you have to tell the Exim daemon to reload it. This is done by sending the daemon a `SIGHUP` signal. All other Exim processes are short-lived, so as new ones start up after the change, they pick up the new configuration.

For very simple installations, it may be possible to include all the configuration data within the runtime configuration file. A minimal usable configuration of this type is shown in the next chapter, in the section "A Minimal Usable Configuration File." Normally, however, the runtime configuration refers to auxiliary data, which can be in ordinary files, or in databases such as NIS or LDAP. Common examples are the system alias file (usually called */etc/aliases*) and users' *.forward* files. Files or databases can also be used for lists of hosts, domains, or addresses that are to be handled in some special way and that are too long to conveniently include within the configuration file itself. Data from such sources is read afresh every time it is needed, so updates take immediate effect and there is no need to send a `SIGHUP` signal to the daemon.

* BSD-based systems tend to use */usr/sbin/sendmail*, whereas Solaris uses */usr/lib/sendmail*. Different MUAs have different defaults, so some administrators set both paths to cater for both kinds.

The simplest item that is found in the runtime configuration file is an option set to a fixed string. For example, the following line:

```
    qualify_domain = example.com
```

specifies that addresses containing only a local part and no domain are to be turned into complete addresses ("qualified") by appending *@example.com.*[*] Each such setting appears on a line by itself. For many option settings, fixed data suffices, but Exim also provides ways for you to supply data that is re-evaluated and modified every time it is used. Examples and explanations of this feature are introduced later in this chapter.

How Exim Delivers Messages

Exim's configuration determines how it processes addresses; this processing involves finding information about the destinations of a message and how to transport it to those destinations. In this and the following sections, we discuss how the configuration that you set up controls what happens.

There are many different ways an address can be processed. For example, looking up a domain in the DNS involves a completely different way of processing from looking up a local part in an alias file, and delivering a message using SMTP over TCP/IP has very little in common with appending it to a mailbox file. There are separate blocks of code in Exim for doing the different kinds of processing, and each is separately and independently configurable. The word *driver* is used as the general term for one of these code blocks. In many cases, when you specify that a particular driver is to be used, you need only give one or two parameters for it. However, most drivers have a number of other options whose defaults can be changed to vary their behavior.

There are four different kinds of drivers. Three of them are concerned with handling addresses and delivering messages, and are called *directors*, *routers*, and *transports*. The fourth kind of driver handles SMTP authentication and is described in Chapter 15, *Authentication, Encryption, and Other SMTP Processing*.

Transports are the components of Exim that actually deliver messages by writing them to files, or to pipes, or over SMTP connections. Directors and routers are very similar in that their job is to process addresses and decide what deliveries are to take place. The difference between them is in the kinds of address that they

[*] Unqualified addresses are accepted only from local processes, or from certain designated remote hosts.

handle; directors handle local addresses and routers handle remote addresses. As Exim has evolved, the original differences in concept between directors and routers have diminished, and it may come about that they are merged in some future release. For the moment, however, a distinction remains.

Before going into more detail, we take a brief look at the way drivers are used as a message makes its way through the system. Exim has to decide whether each address is to be delivered on the local host or to a remote one, then it has to choose the right form of transport for each address (appending to a user's mailbox, for instance, or connecting to another host via SMTP), and finally it has to invoke those transports. For example, in a typical configuration, a message addressed to *bug_reports@exim.example*, where *exim.example* is a local domain, might be handled like this:

1. The first driver in the configuration is a director that handles system aliases; this tells Exim to check the */etc/aliases* file. Here it finds that the local part *bug_reports* is indeed an alias, and that it resolves to two other addresses: the local address *brutus@exim.example*, and the remote address *julia@helper-sys.org.example*. Further drivers must be invoked to handle each of these new recipients.
2. Later in the configuration is a director that recognizes local users like *brutus*, and it arranges for Exim to run a transport called *appendfile*, which adds a copy of the message to Brutus' mailbox. The actual delivery does not take place until after Exim has worked out how to handle all the addresses.
3. For the other recipient, Exim runs a router that looks up the domain *helper-sys.org.example* in the DNS, and finds the IP address of the remote host to which the message should be sent. It then arranges for Exim to run the *smtp* transport in order to do the delivery.

This example has introduced several of the most commonly used drivers. Later in this chapter, we work through a similar example in much more detail. The individual drivers are described in their own sections in later chapters; here is an alphabetical list of them:

aliasfile
: A director that expands aliases into one or more different addresses.

appendfile
: A transport that writes messages to local files.

autoreply
: A transport that generates automatic replies to messages.

domainlist
: A router that routes remote domains using locally supplied information.

forwardfile
: A director that handles users' *.forward* files and Exim filter files.

ipliteral
: A router that handles "IP literal" addresses such as *user@[192.168.5.6]*. These are relics of the early Internet that are no longer in common use.

lmtp
: A transport that delivers messages to external processes using the LMTP protocol.*

localuser
: A director that recognizes local usernames.

lookuphost
: A router that looks up remote domains in the DNS.

pipe
: A transport that passes messages to external processes via pipes.

queryprogram
: A router that runs an external program in order to route a domain.

smartuser
: A director that accepts any address; it is used as a "catchall."

smtp
: A transport that writes messages to other hosts over TCP/IP connections, using either SMTP or LMTP.

The configuration may refer to the same driver code more than once, but with different options, in order to create multiple instances of the same driver type. Each driver instance is given an identifying name in the configuration file, for use in logging and for reference from other drivers.

Local and Remote Addresses

There are two distinct types of mail address: those for which the local part is used when deciding how to deliver the message, and those for which only the domain is relevant. Typically, when a domain refers to a remote host, the local part of the address plays no part in the routing process, but if the domain is the name of the local host, the local part is usually used in determining where to deliver the message. This is not a hard and fast rule (a small company might accept mail for any

* LMTP (RFC 2033) is a variation of SMTP that is designed for passing messages between local processes.

local part in a single mailbox), but it forms the basis of the distinction between directors and routers.

The first thing Exim does when processing an address is to determine whether it should be handled by the directors or by the routers. An Exim configuration normally contains definitions of a number of directors and at least one router, though there may be any number of either. If the domain is listed in the configuration as a *local domain*, the address is processed by the directors and is called a *local address*. Otherwise it is processed by the routers and is called a *remote address*.

Exim decides whether a domain is local by checking the `local_domains` option, which contains a colon-separated list of patterns. If it is not set, the name of the local host is used as the only local domain. Otherwise, it may contain various types of patterns, of which the most common are shown in this example:

```
local_domains = tiber.rivers.example:\
                *.cities.example:\
                dbm;/usr/exim/domains
```

The first item in the list is a single domain name, *tiber.rivers.example*, while the second is a simple pattern, matching all domains that end in *.cities.example*.[*] The third item is a reference to an external file, */usr/exim/domains*, which is a DBM-keyed file. This type of item is useful when a host is handling a very large number of local domains. We discuss DBM files and this kind of lookup item in more detail later.

Notice the use of backslashes for continuing the option value over several lines. This is a general feature of Exim's configuration file; any line can be continued in this way. Whitespace at the start of continuation lines is ignored.[†]

Processing an Address

After it has decided whether an address is local or remote, Exim offers it to each configured director or router (as appropriate) in turn, in the order in which they are defined, until one of them is able to deal with it. The order in which directors and routers are defined in the configuration file is therefore important. The process of directing a local address is illustrated in Figure 3-2; a similar process happens using the routers for a remote address.

A director that successfully handles an address may add that address to a queue for a particular transport. Alternatively, it may generate one or more "child"

* More complicated patterns can be given in the form of regular expressions.

† In versions of Exim prior to 3.14, this continuation mechanism is available only in macro definitions, rewriting rules, and option settings where the value is given enclosed in double quotes. Thus, the earlier example would have to be quoted if used in an earlier version.

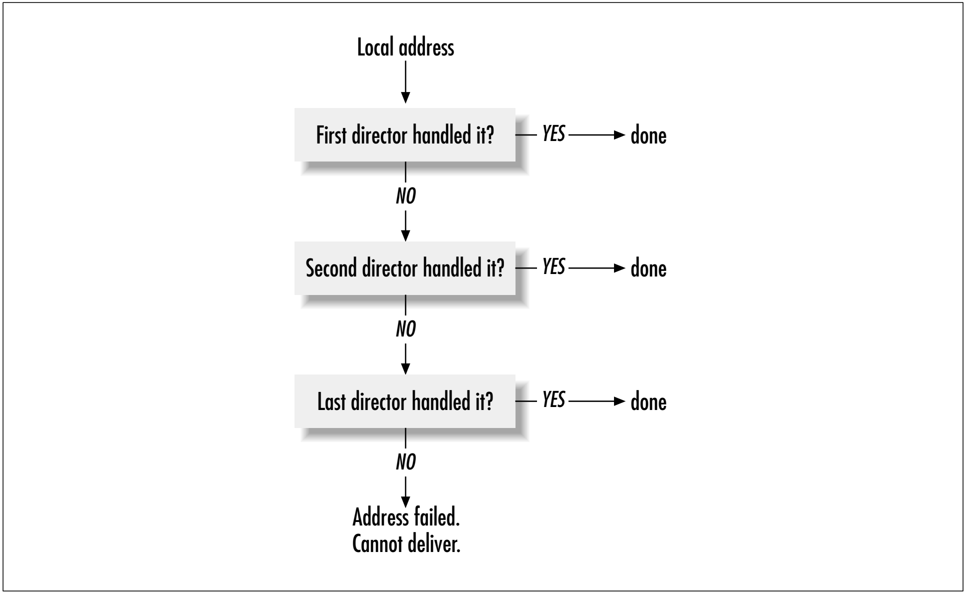

Local address
First director handled it?
YES
done
NO
Second director handled it?
YES
done
NO
Last director handled it?
YES
done
NO
Address failed.
Cannot deliver.

Figure 3-2. Directing a local address

addresses that are added to the message's address list and processed in their own right, with the original address no longer playing any part. This is what happens when a local part matches an entry in an alias list, or when a user's *.forward* file is activated.

A successful router, on the other hand, can only add the address to a queue for a transport, or modify the domain and pass it on to the next router. It cannot generate "child" addresses. When a director or a router cannot handle an address, it is said to *decline*. If every director or router declines, the address cannot be handled at all, and delivery fails.

The way addresses are handled by directors and routers is illustrated in Figure 3-3. (The line labeled "local after all" is a special case that is discussed in the section "Remote Address Becoming Local," later in this chapter.) All the addresses in a message, and any that are generated from them (for example, by aliasing), are processed by the directors and routers before any deliveries take place from the transport queues. Any router or director can queue an address for any transport; directors are not restricted to local transports, nor routers to remote ones.

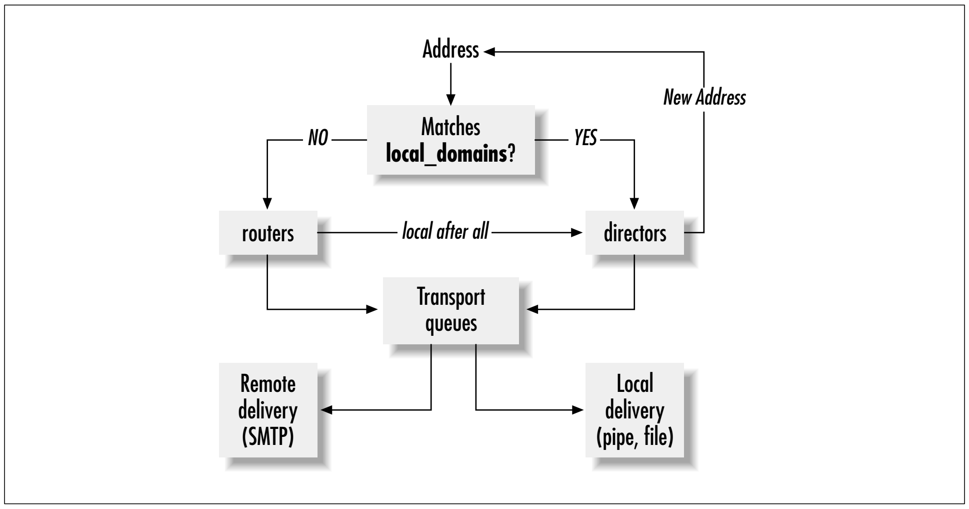


Figure 3-3. Routing and directing

A Simple Example

To help clarify the mechanisms described earlier, an example of a simple message delivery is presented here. The scenario is a host called *simple.example*, where the hostname is the only local mail domain. The host is using a simple Exim configuration file that supports aliases, user-forward files, delivery to local users' mailboxes, and remote SMTP delivery. The relevant portions of the configuration are quoted here. Suppose a user of this host has sent a message addressed to one local and one remote recipient:

```
postmaster@simple.example
friend@another.example
```

At the start of delivery, Exim's list of addresses to process is initialized with the two original recipients, and its first job is to work through this list, deciding what to do for each address. For *postmaster@simple.example*, the domain is local, so it is passed to the first defined director, whose configuration is as follows:

```
system_aliases:
  driver = aliasfile
  file = /etc/aliases
  search_type = lsearch
```

The first line, terminated by a colon, is the name for this particular director instance, chosen by the system administrator. Each driver of a particular type (director, router, or transport) must have a distinct name. However, names of driver instances can be the same as the names of the drivers themselves; you can have the following:

```
aliasfile:
  driver = aliasfile
  file = /etc/aliases
  search_type = lsearch
```

if you want to, but some people find this usage confusing. The second configuration line specifies which kind of director this is (or, to put it another way, it chooses which block of director code to run), and the remaining two lines are options for the director.

The **aliasfile** director handles an address by looking up the local part in an alias list, and the options control how the lookup is done. In this case, the list is in the file */etc/aliases*, and a linear search ("lsearch") is required. This expects each line of the file to contain an alias name, optionally terminated by a colon, followed by the list of replacement addresses for the alias, which may be continued onto subsequent lines by starting them with whitespace. A comma is used to separate addresses in the list. For example:

```
root:         postmaster@simple.example,
              herb@simple.example
postmaster:   simon@simple.example
```

Notice that the first line specifies that *root* is an alias for *postmaster*, which itself is an alias. This is a common practice, and works exactly as you might expect.* The **aliasfile** director reads through this file and finds the entry for *postmaster*, so it adds a new address, *simon@simple.example*, to the list of addresses to process, and returns a code that indicates success, meaning that *postmaster@simple.example* has been completely processed. The list of pending addresses now contains the following:

```
simon@simple.example
friend@another.example
```

Exim proceeds to tackle *simon@simple.example*,† which is another local address, so again it is offered to the **system_aliases** director. This time, however, there is no match in */etc/aliases*, so the director cannot handle the address. It returns a code indicating "decline," which causes Exim to offer the address to the next director, whose configuration is as follows:

```
userforward:
  driver = forwardfile
  file = .forward
```

The job of a **forwardfile** director is to check for the existence of files containing lists of forwarding addresses. This instance is configured to look for *.forward* files in users' home directories. First of all, it has to check that the local part of the

* See the section "Directing Loops," later in this chapter, for a discussion of how it might go wrong.

† In practice, it might not actually happen in this order.

address corresponds to a user login name.[*] If there is no matching user, the director declines, but if *simon* is in fact a user of the host, the director goes on to check the existence of the given file.

If the file is defined using a relative pathname, as shown earlier, it is sought in the user's home directory. Because home directories are often NFS-mounted, Exim first checks that the directory is available before trying to open the file so that the absence of the directory is not mistakenly interpreted as the absence of the file.[†]

If *simon* has a *.forward* file, its contents are a list of forwarding addresses and other types of items, as described in the section "Items in Alias and Forward Lists," in Chapter 7, *The Directors*. The addresses are added to the list of addresses to process, the **userforward** director returns a code indicating success, and the new addresses are eventually processed independently.

If *simon* does not have a *.forward* file, the director declines, and *simon@simple.example* is offered to the third director in the configuration:

```
localuser:
  driver = localuser
  transport = local_delivery
```

The job of **localuser** is to check whether the local part of the address corresponds to a user login name. In this configuration, this check has already been done by the previous director. This is quite a common occurrence, so Exim keeps a cache of the most recently looked-up name to avoid wasteful repetition. If *simon* were not a local user, the director would decline, and as there are no more directors in the configuration, the address would fail. It would be placed on a list of failed addresses and used to generate a bounce message at the end of the delivery attempt.

When the local user does exist, the director succeeds, and it places the address on a queue for the **local_delivery** transport, attaching to it the uid, gid, and home directory that it looked up. That is all that happens at this stage; no actual delivery takes place until later. The processing of *postmaster@simple.example* is illustrated in Figure 3-4, where the ovals represent sources of information, and the rectangles represent drivers.

There is still one address to process: *friend@another.example*. Its domain is not a local one, so it is processed by routers rather than by directors. Exim offers it to the first router:

---

[*] It does this by calling the system function `getpwnam()` rather than looking at */etc/passwd* directly, so that users defined by other means (such as NIS) are recognized.

[†] In an automounted environment, the directory check causes an automount to occur.

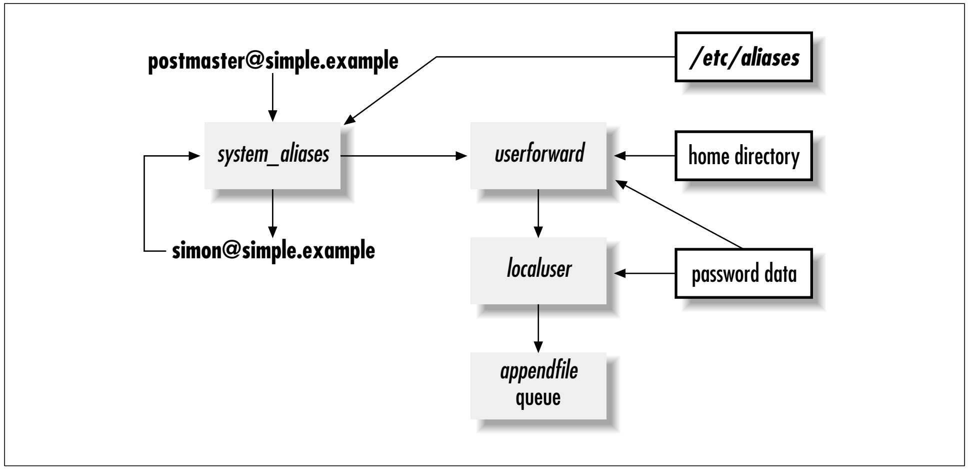


Figure 3-4. Directing example

```
lookuphost:
  driver = lookuphost
  transport = remote_smtp
```

This is in fact the only router in this simple configuration, so if it declines, the address fails. The job of **lookuphost** is to obtain a list of remote hosts for the domain of an address, and in its normal configuration (as shown earlier), it does this by looking up the domain in the DNS using MX and address records, as described in the section "DNS Records Used for Mail Routing," in Chapter 2, *How Internet Mail Works*. When it is successful, it ends up with an ordered list of hosts and their IP addresses. It puts the mail address on a queue for the **remote_smtp** transport, attaching the host list. In our example, if the MX and address records were the following:

```
another.example.         MX   6   mail-2.another.example.
another.example.         MX   4   mail-1.another.example.
mail-1.another.example.  A        192.168.34.67
mail-2.another.example.  A        192.168.88.32
```

then the list of hosts to be passed with the address to **remote_smtp** would be:

```
mail-1.another.example  192.168.34.67
mail-2.another.example  192.168.88.32
```

Any hosts that have the same MX preference value are sorted into a random order. The processing of *friend@another.example* is illustrated in Figure 3-5.

There are now no more unprocessed addresses, so the directing and routing phase of the delivery process is complete, and Exim moves on to do the actual deliveries by running the transports that have been set up. Local transports are run first; in our example, there is one local delivery setup for the address *simon@simple.example*, using the **local_delivery** transport. This was specified by a **localuser** director

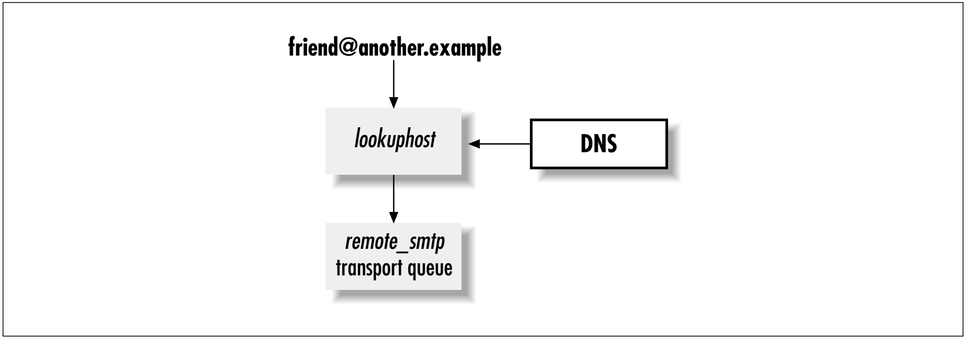


Figure 3-5. Routing example

that handled the address. The transport is configured thus:

```
local_delivery;
  driver = appendfile
  file = /var/mail/$local_part
  delivery_date_add
  envelope_to_add
  return_path_add
```

This uses the **appendfile** driver, which adds a copy of the message to the end of a mailbox file in conventional Unix format when configured in this way.*

The name of the file is given by the `file` option. Its value, with an embedded dollar character, is different from the option settings that we have met so far, which have all been fixed values. Much of the flexibility of Exim's configuration comes from the use of option settings where the specified strings are changed each time they are used. This process is called *string expansion*, and we'll see it in many examples throughout this book. A complete description of all the expansion features is given in Chapter 17, *String Expansion.*

The simplest change that can be made to a string is the insertion of a variable value, and this is what is happening in the earlier example. Exim replaces teh substring `$local_part` by the local part of the address that is being delivered, so the file that is actually used is */var/mail/simon.* The remaining three options request the addition of three generally useful header lines as the message is written:

Delivery-Date:
  A header that records the date and time of delivery, for example:

```
Delivery-Date: Fri, 31 Dec 1999 23:59:59 +0000
```

* Other configurations (see the section "The appendfile Transport," in Chapter 9, *The Transports*) support different formats.

Envelope-To:
: A header that records the original recipient address (the "envelope to" address) that caused the delivery; in this example it would be:

```
    Envelope-To: postmaster@simple.example
```

Preserving this address is useful in case it does not appear in the *To:* or *Cc:* headers.

Return-Path:
: A header that records the sender from the message's envelope, for example:

```
    Return-Path: <user@simple.example>
```

For bounce messages that have no sender, it looks like this:

```
    Return-Path: <>
```

Local deliveries are always run sequentially in separate processes that change their user identity to some specific value. In this case, the user ID (uid) and group ID (gid) of the local user were passed to the transport by the **localuser** director, so these are used. The delivery subprocess is therefore running "as the user" when it accesses the mailbox.*

When the subprocess has finished, there are no more local deliveries, so Exim proceeds to the remote ones. Before it does so, it gives up its root privilege permanently, and runs as the Exim user if a uid and gid for Exim have been defined in the configuration (either at build time or at runtime). This is the recommended way to run Exim.

There is one remote delivery, for *friend@another.example*, which was set up by the **lookuphost** router to use the **remote_smtp** transport:

```
remote_smtp:
  driver = smtp
```

There are no option settings here beyond the one that selects the type of transport, because the list of hosts was obtained by the **lookuphost** router and passed to the transport along with the address. The parameters of the outgoing SMTP call (for example, the timeouts) can be changed by other options, but in this case we accept all the defaults. The **smtp** transport tries to make an SMTP connection to each host in turn. If all goes well, a connection is made to one of them, and the message is transferred.

There are now no more deliveries to be done, and all the recipients have been successfully handled, so at this point Exim can delete the message files on its

* The use of different uids and gids in Exim is discussed in the section "Security Issues," in Chapter 19, *Miscellany.*

spool and log the fact that this message has been delivered. The delivery process then exits.

The fragments of configuration file used in this example have been shown in the order in which they are used during delivery. The actual configuration file defines the transports first, followed by the directors, and finally the routers. The transports come first, so that when Exim is reading the file, they are defined before the director and router configurations that refer to them. In the following chapter, we show a complete configuration file.

Complications While Directing and Routing

Things do not always go as smoothly as described in the simple example. These are some of the more common complications that can be encountered when directing or routing an address.

Duplicate Addresses

Duplicate addresses are a complication that Exim may have to handle, either because the sender of the message specified the same address more than once, or because aliasing or forwarding duplicated an existing recipient address. For any given address, only a single delivery takes place, except when the duplicates are pipe commands. If one user is forwarding to another, and a message is sent to both of them, only a single copy is delivered. If, on the other hand, two different users set up their *.forward* files to pipe to */usr/bin/vacation* (for example), a message that is sent to both of them runs the vacation program twice, once as each user.

Missing Data

Sometimes, a director or router is unable to determine whether it can handle an address. For example, if the administrator has misspelled the name of an alias file, or if it has been accidentally deleted, an **aliasfile** director cannot operate. Timeouts can occur when a router queries the DNS, and both routers and directors can refer to databases that may at times be offline. In these situations, the director or router returns a code indicating "defer" to the main part of Exim, and the address is neither delivered nor bounced, but left on the spool for another delivery attempt at a later time. The control of retry times is described in Chapter 12, *Delivery Errors and Retrying*. If the error condition is felt to be sufficiently serious, the message is "frozen," which means that queue runner processes will not try to deliver it. As frozen messages are highlighted in queue listings, this also serves to bring it to the administrator's attention.

Directing Loops

When an **aliasfile** or **forwardfile** director handles an address, the new addresses that it generates are each processed afresh, just like the original recipient addresses.* This means that one alias can refer to another, as in the example we showed earlier:

```
root:        postmaster@simple.example
postmaster:  simon@simple.example
```

However, it opens up the possibility of directing loops. To prevent this, Exim automatically skips a director if the address it is handling has a "parent" address that was processed by that director. Consider the following broken alias file:

```
chicken:    egg@simple.example
egg:        chicken@simple.example
```

This director turns a message addressed to *chicken@simple.example* into *egg@simple.example*, and then turns it back into *chicken@simple.example* the next time through. However, on the third pass, Exim notices that the address was previously processed by the director, so it is skipped and the next director is called. The chances are that the resulting delivery or bounce are not what was intended, but at least the loop is broken.

Remote Address Becoming Local

It sometimes turns out that when a router is processing an address, it discovers that the domain is a local domain after all. This can happen if the domain was originally given in an abbreviated form (for example, as in the address *brutus@rome*), because DNS lookups are commonly configured to expand single-component names into the full form, within the local encompassing domain. If routing changes the domain name, and the result is a local domain, the address is automatically passed from the router to the directors.

* This is the normal practice; there are occasions when it is not wanted, and there is an option, `new_director`, that can be used to disable it.

Remote Address Routing to the Local Host

After Exim has routed a remote address, it checks to see whether the first host on the list of hosts to which the message could be sent is the local host. Usually, this indicates some kind of configuration error, and by default Exim treats it as such. However, there are types of configuration where it is legitimate, and for these cases the `self` option can be used to pass such addresses from the router to the directors.*

Complications During Delivery

A successful routing process for a remote address discovers a list of hosts to which it can be sent, but it cannot check the local part of the address. The most common permanent error during a remote delivery is "unknown user," which is given in response to an SMTP `RCPT` command. Responsibility for the message remains with the sending host, which must return a bounce message to the sender.

Not all receiving hosts behave like this; some accept any local part (in their local domain) during the SMTP dialog, and do the check later. By this time, responsibility for the message has been passed, so it is the receiving host that has to generate the bounce. When Exim is a receiving host, it can be configured to act in either manner, depending on the setting of `receiver_verify` and related options (see the section "Verifying Recipient Addresses," in Chapter 13, *Message Reception and Policy Controls*).

There are other reasons a remote host might permanently refuse a message, and in addition, there are many common temporary errors, such as the inability to contact a host. These cause a message to remain on the spool for later delivery.

In contrast to routing, directors for local addresses normally check local parts, so any "unknown user" errors happen at directing time. The only problems a local transport is likely to encounter are errors in the actual copying of the message. The most common is a full mailbox; Exim respects system quotas and can be configured to impose its own quotas (see the section "Mailbox Quotas," in Chapter 9). A quota failure leaves the message on the spool for later delivery.

The runtime configuration contains a set of retry rules (see Chapter 12) that specify how often, and for how long, Exim is to go on trying to deliver messages that are suffering temporary failures. The rules can specify different behaviors for different kinds of error.

* See, for example, the section "Mixed Local/Remote Domains," in Chapter 5, *Extending the Delivery Configuration.*

Complications After Delivery

When all delivery attempts for a message are complete, a delivery process has two final tasks. If any deliveries suffered temporary errors, or if any deliveries succeeded after previous temporary errors, the delivery process has to update the retry hints database. This work is saved up for the end of delivery so that the process opens the *hints* database for updating only once at most, and for as short a time as possible. If the updating should fail, the new hint information is lost, but previous hint information remains. In practice, except in exceptional circumstances such as a power loss, hint information is rarely lost.

Finally, unsuccessful delivery may cause a message to be sent to the sender. If any addresses failed, a single bounce message is generated that contains information about all of them. If any addresses were deferred, and have been delayed for more than a certain time (see the section "Delay Warning Messages," in Chapter 19), a warning message may be sent.

Exim sends such messages by calling itself in a subprocess. Failure to create a bounce message causes Exim to write to its panic log and immediately exit. This has the effect of leaving the message on the spool so that there will be another delivery attempt, and presumably another attempt at sending the bounce message when the delivery fails again. Failure to create a warning message, on the other hand, is not treated as serious. Another attempt to send it is made when the original message is processed again.

Use of Transports by Directors and Routers

In the simple example we have been considering, the **localuser** director and the **lookuphost** router include the `transport` option, referring to the **local_delivery** and **remote_smtp** transports, respectively, whereas the other directors do not have any transport settings. A transport is required for any router or director that actually sets up a message delivery to determine how the delivery should be done. When a director is just changing the delivery address by aliasing or forwarding, a transport is not required because no delivery is being set up at that stage.

Depending on their configurations, some directors and routers require a transport setting, and some require there is not a transport setting. Exim detects an incorrect configuration when the configuration file is read. In other cases, the director or router may behave differently, depending on whether or not a transport is supplied. These variations are explained in the detailed descriptions of the directors and routers (see Chapter 7 and Chapter 8, *The Routers*).

Two directors, **aliasfile** and **forwardfile**, have additional options for special-purpose transports. These directors can deliver a message to a specific file, or to a pipe associated with a given command. For example, a line in an alias file of the form:

```
majordomo:  |/usr/mail/majordomo ...
```

specifies that a message addressed to the local part *majordomo* is to be passed via a pipe to a process running the command:

```
/usr/mail/majordomo ...
```

The other entries in the alias file may just be changing delivery addresses, and therefore may not require a transport. However, this line is setting up a delivery, and so a transport is required. We can add to the **system_aliases** director configuration the following line, which in our example runs the **aliasfile** director:

```
address_pipe_transport = alias_pipe
```

This tells Exim which transport to run when a pipe is specified in the alias file. The transport itself is very simple:

```
alias_pipe:
  transport = pipe
  ignore_status
  return_output
```

A **pipe** transport runs a given command in a new process, and passes the message to it using a pipe for its standard input. In this example, the command is provided by the alias file, so the transport does not need to define it.* Setting `ignore_status` tells Exim to ignore the status returned by the command; without this, any value other than zero is treated as an error, causing the delivery to fail and a bounce message to be returned to the sender.

Setting `return_output` changes what happens if the command produces output on its standard output or standard error streams. By default, such output is discarded, but if `return_output` is set, the production of such output is treated as an error, and the output itself is returned to the sender in the bounce message.

There is one piece of information that the **pipe** transport needs that we have not yet given, and that is the uid and gid under which it should run the command. When a pipe is triggered by an entry in a user's *.forward* file, the user's identity is assumed by default, but when an alias file is used, as it is here, there is no default.

* If the **pipe** transport is run directly from a director or router, the command to be run is defined using its `command` option.

The `user` (and, optionally, `group`) option can appear in either the director or the transport's configuration, so the transport could become:[*]

```
alias_pipe:
  transport = pipe
  ignore_status
  return_output
  user = majordom
```

In addition to delivery to pipes, alias files and forward files may also specify specific files into which messages are to be delivered. For example, if user *caesar* has a *.forward* containing:

```
caesar@another.domain.example, /home/caesar/mail-archive
```

it requests delivery to another mail address, and also into the named file, which is a delivery that needs a transport. To support this feature, the **userforward** director could contain:

```
address_file_transport = address_file
```

This tells Exim which transport to run when a filename is specified instead of an address in a forward file. The transport itself is even more simple than the **pipe** transport:

```
address_file:
  driver = appendfile
```

The filename comes from the forward file, and all other options are defaulted.

An alias or forward file may contain both of these kinds of entries, thus requiring both `address_pipe_transport` and `address_file_transport` to be given on a single director. These options are used for these very specific purposes only, and should not be confused with the generic `transport` option that applies to all directors and routers.

[*] This assumes that all the pipes specified in the alias file are to be run under the same uid. If there are several instances that require different user identities, an expansion string can be used to select the correct uid, but that is too advanced for the discussion here.

4

Exim Operations Overview

The previous chapter used some fragments of a simple Exim configuration file to show how it goes about delivering a message. Later chapters go into more detail about the various options that can be used to set up configurations that can handle many different circumstances. However, if you have just installed Exim, or if you have inherited responsibility for an Exim system from somebody else, you most likely want to know a little bit about the basic operational aspects. This chapter is an introductory overview; the features that are described reappear later in more detailed discussions, and Chapter 21, *Administering Exim*, covers Exim administration in more detail.

How Exim Identifies Messages

Each message that Exim handles is given a unique *message ID* when it is received. The ID is 16 characters long, and consists of three parts, separated by hyphens. For example:

```
    11uNWX-0004fP-00
```

Each part is actually a number, encoded in base 62. The first is the time that the message started to be received, and the second is the ID of the process (the pid) that received the message. The third part is used to distinguish between messages that are received by the same process in the same second. It is almost always 00.

The uniqueness of Exim's message IDs relies on the fact that Unix process IDs are used cyclically, so in practice there is no chance of the same process ID being reused within one second. For most installations, uniqueness is required only within a single host, and the scheme just described suffices. However, in some cluster configurations, it is useful to ensure that message IDs are unique within the cluster. For example, suppose two hosts are providing identical gateway or hubbing services for some domain, and one of the processors has a catastrophic failure. If its disk can be attached to the other processor, and the message IDs are unique across both systems, spooled message files can simply be moved into the survivor's spool directory.

Uniqueness across several hosts can be ensured by assigning each host a number in the range 0–255, and specifying it in each Exim configuration. For example:

```
localhost_number = 4
```

When this option is set, the third part of the message ID is no longer a simple sequence number. Instead, it is computed as:

```
sequence number * 256 + host number
```

For example, in the following message ID:

```
11vHQS-0006ZD-4C
```

the number `4C` is 260 in decimal, which is `256 * 1 + 4`, so this message ID was generated on host number 4 for the second message received by some process within one second. In the most common case, when the sequence number is zero, the final part of the message ID is just the host number.*

Watching Exim at Work

As a new administrator of an MTA, the first questions you should ask are:

- How do I find out what messages are on the queue?
- How do I find out what the MTA has been doing?

Exim can output a list of its queue in a number of ways, which are detailed in the section "Watching Exim's Queue," in Chapter 20, *Command-Line Interface to Exim*. The most basic is the *-bp* command-line option. This option is compatible with Sendmail, though the output is specific to Exim:†

```
$ exim -bp
25m  2.9K  0t5C6f-0000c8-00 <caesar@rome.example>
           brutus@rome.example
```

* In this type of configuration, the maximum sequence number is 14. If more than 14 messages are received by one process within one second, a delay of one second is imposed before reading the next message, in order to allow the clock to tick.

† In examples of commands that are run from the shell, the input is shown in boldface type.

This shows that there's just one message, from *caesar@rome.example* to *brutus@rome.example*, which is 2.9 KB in size, and has been on the queue for 25 minutes. Exim also outputs the same information if it is called under the name *mailq*, which is a fairly common convention.*

Exim logs every action it takes in its main log file. A log line is written whenever a message arrives and whenever a delivery succeeds or fails. The name of the log file depends on the configuration, with two common choices being */var/spool/exim/log/mainlog* or */var/log/exim_mainlog*.† If you have access to an X Window server, you can run the *eximon* utility, which displays a "tail" of the main log in a window (see the section "The Exim Monitor," in Chapter 21). The entries that Exim writes to the log are described in detail in the section "Log Files," in Chapter 21.

Exim uses two additional log files that are in the same directory as the main log. One is called *rejectlog*; it records details of messages that have been rejected for reasons of policy. The other is called *paniclog*; this is used when Exim encounters some disaster that it can't handle. The paniclog should normally be empty; it is a good idea to set up some automatic monitoring to let you know if something has been written to it, because that usually indicates there has been an incident that warrants investigation.

The Runtime Configuration File

Exim's runtime configuration is held in a single text file that you can modify with your favorite text editor. If you make a change, newly started Exim processes will immediately pick up the new file, but the daemon process will not. You have to tell the daemon to reread its configuration, and this is done in the traditional Unix way, by sending it a HUP signal. The process number of the daemon is stored in Exim's spool directory, so that you can do this by running (as *root* or *exim*) the following command:

```
kill -HUP `cat /var/spool/exim/exim-daemon.pid`
```

On receiving a HUP signal, the daemon closes itself down, and then restarts in a new process, thereby picking up the new configuration.

* Many operating systems are set up with the *mailq* command as a symbolic link to *sendmail*; if this in turn has been linked to *exim*, the *mailq* command will "just work."

† It is possible to configure Exim to use *syslog* instead, but this has several disadvantages.

Layout of the Configuration File

The runtime configuration file is divided into seven different sections, as shown in Figure 4-1. It consists of the following sections:

Main section
: General option settings and input controls

Transport section
: The configuration for the transports

Director section
: The configuration for the directors

Router section
: The configuration for the routers

Retry section
: The retry rules for specifying how often Exim is to retry temporarily failing addresses (see Chapter 12, *Delivery Errors and Retrying*)

Rewriting section
: The global address rewriting rules (see Chapter 14, *Rewriting Addresses*)

Authenticator section
: The configuration for the SMTP authenticators (see the section "SMTP Authentication," in Chapter 15, *Authentication, Encryption, and Other SMTP Processing*)

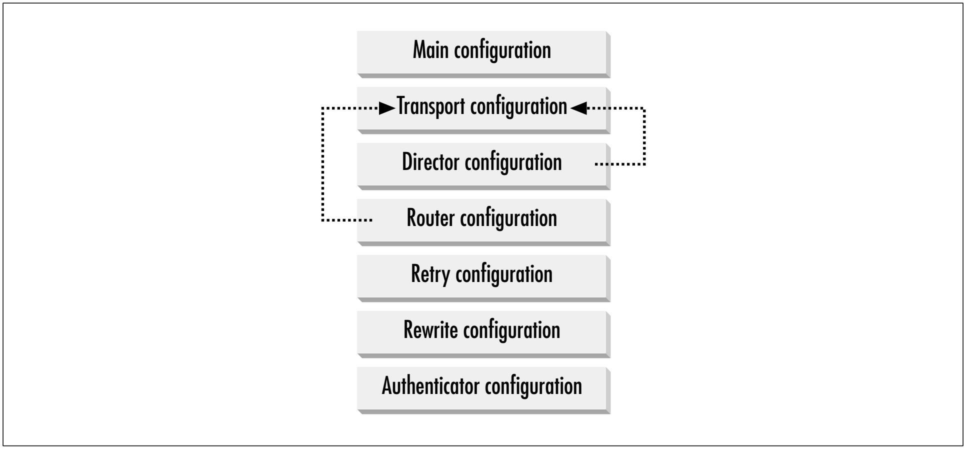


Figure 4-1. Runtime configuration file

The arrows in the figure indicate that the drivers defined in the directors and routers sections refer back to the transports that are defined in the transports section. We saw an example of this in the previous chapter, where the **lookuphost** router referred to the **remote_smtp** transport:

```
lookuphost:
  driver = lookuphost
  transport = remote_smtp
```

In the actual file, the separators between the sections are lines containing just the word `end`. Sections that are not needed may be empty, and if they occur at the end of the file, they can be completely omitted. This means that a completely empty file is, in fact, a valid configuration file, but it would not be much use because no way to deliver messages is defined.

The retry and rewriting configuration sections each contain lines in a format that is unique to the section, and we discuss these in later chapters. The remaining sections contain option settings in the form *name*=*value*, one per line. Except when we are discussing a specific driver, unqualified references to options always refer to one of the options in the main configuration section.

A Minimal Usable Configuration File

The simplest complete configuration that is capable of delivering both local and remote mail is as follows:

```
# Main configuration: all defaults taken

end

# Transports: SMTP and local mailboxes

remote_smtp:
  driver = smtp

local_delivery:
  driver = appendfile
  file = /var/mail/$local_part

end

# Directors: local user mailbox only

localuser:
  driver = localuser
  transport = local_delivery

end
```

```
# Routers: standard DNS routing

lookuphost:
  driver = lookuphost
  transport = remote_smtp
```

Lines beginning with # characters are comments, which are ignored by Exim. This example is cut down from the default configuration, and is even simpler in its handling of local domains than the case we considered in the previous chapter; it does not support aliasing or forwarding. Because there are no retry rules in this configuration, messages that suffer temporary delivery failures will be returned to their senders without any retries, so this would not be a very good example to use for real.

Notice that, although there are no settings in the main part of the configuration (so that default values are used for all the options), the `end` line is still required to mark the end of the section.

Option Setting Syntax

We've already seen a number of examples of option settings. Each one is on a line by itself, and they can always be in the form *`name`*`=`*`value`*. For those that are on/off switches (Boolean options), other forms are also permitted. The name on its own turns the option on, whereas the name preceded by `no_` or `not_` turns it off. These settings are all equivalent:

```
sender_verify
sender_verify = true
sender_verify = yes
```

So are these:

```
no_sender_verify
not_sender_verify
sender_verify = false
sender_verify = no
```

You do not have to use quote marks for option values that are text strings, but if you do, any backslashes in the strings are interpreted specially.* For example, the sequence `\n` in a quoted string is converted into a linefeed character. This feature is not needed very often.

Some options specify a time interval; for example, the timeout period for an SMTP connection. A time interval is specified as a number followed by one of the letters

* Exim recognizes only double-quote characters for this purpose.

`w` (week), `d` (day), `h` (hour), `m` (minute), or `s` (second). You can combine several of these to make up one value. For example, the following:

```
connect_timeout = 4m30s
```

specifies a time interval of 4 minutes and 30 seconds.

Macros in the Configuration File

For more complicated configuration files, it may be helpful to make use of the simple macro facility. If a line in the main part of the configuration (that is, before the first `end` line) begins with an uppercase letter, it is taken as a macro definition, of the form:

```
name = rest of line
```

The name must consist of letters, digits, and underscores, and need not all be in uppercase, though that is recommended. The rest of the logical line is the replacement text, and has leading and trailing whitespace removed. Quotes are not removed.

Once a macro is defined, all subsequent lines in the file are scanned for the macro name; if there are several macros, the line is scanned for each in turn, in the order in which they are defined. The replacement text is not rescanned for the current macro, though it will be for subsequently defined macros. For this reason, a macro name may not contain the name of a previously defined macro as a substring. You could, for example, define the following:

```
ABCD_XYZ = something
ABCD = something
```

but putting those definitions in the opposite order would provoke a configuration error.

As an example of macro usage, suppose you have lots of local domains, but they fall into three different categories. You could set up the following:

```
LOCAL1 = domain1:domain2
LOCAL2 = domain3:domain4
LOCAL3 = dbm;/list/of/other/domains

local_domains = LOCAL1:LOCAL2:LOCAL3
```

and use the `domains` option on appropriate directors to handle each set of domains differently. This avoids having to list each domain in more than one place.* The values of macros can be overridden by the *-D* command-line option (see the section "Configuration Overrides," in Chapter 20).

* However, there may be a difficulty if you are using negated items in the list. This is explained in the section "Negative Items in Lists," in Chapter 18, *Domain, Host, and Address Lists.*

Hiding Configuration Data

The command-line option *-bP* requests Exim to output the value of one or more configuration options. This can be used by any caller of Exim, but some configurations may contain data that should not be generally accessible. For example, a configuration that references a MySQL database or an LDAP server may contain passwords for controlling such access. If any option setting is preceded by the word `hide`, only an admin user is permitted to see its value. For example, if the configuration contains:

```
hide mysql_servers = localhost/usertable/admin/secret
```

an unprivileged user gets this response:

```
$ exim -bP mysql_servers
mysql_servers = <value not displayable>
```

This feature was added to Exim at Release 3.20.

String Expansions

We have already met a simple string expansion in the following setting:

```
file = /var/mail/$local_part
```

for an **appendfile** transport. Expansions are a powerful feature in configuration files. We explain some more of their abilities in examples in subsequent chapters. If you want to know about everything they can do, skip ahead to Chapter 17, *String Expansion*, which has the full story. Meanwhile, remember that whenever you see a `$` character in a configuration setting, it means that the string will change in some way whenever it is expanded for use.

Incorrect syntax in a string expansion is a serious error, and usually causes Exim to give up what it is trying to do; for example, an attempt to deliver a message is deferred if Exim cannot expand a relevant string. However, there are some expansion constructions that deliberately provoke a special kind of error, called a *forced expansion failure*; in a number of such cases, these failures just cause Exim to abandon the activity that uses the string, but otherwise to carry on. For example, a forced expansion failure during an attempt to rewrite an address just abandons the rewriting. Whenever a forced expansion failure has a special effect like this, we'll mention it.

File and Database Lookups

The ability to use data from databases and files in a variety of formats is another powerful feature of Exim's configuration. Earlier, we showed this director for handling traditional alias files:

```
aliasfile:
  driver = aliasfile
  file = /etc/aliases
  search_type = lsearch
```

This looks up data in */etc/aliases* by means of a linear search, but it could equally use an indexed file format such as DBM:

```
aliasfile:
  driver = aliasfile
  file = /etc/aliases.db
  search_type = dbm
```

or, the aliasing data could be held in a database:

```
aliasfile:
  driver = aliasfile
  query = select addresses from aliases where name='$local_part'
  search_type = mysql
```

Each different lookup type is implemented in a different module. Which ones are included in the Exim binary is configured when Exim is built. As far as the main part of Exim is concerned, there is a fixed internal interface (API) to these lookups, and it is unaware of the details of the actual lookup mechanism. However, it does distinguish between the two different kinds of lookup:

Single-key
: Use a single key string to extract data from a file. The key and the file have to be specified.

Query-style
: Access a database using a query written in the query language of the database package.

As well as being configured in options for drivers such as **aliasfile**, lookups can be used in expansion strings to replace part of the string with data that comes from a file or database. They can also be used as a mechanism for managing lists of domains, hosts, or addresses. We encounter examples of these uses throughout the book. Full details of all the lookup types and how they operate are given in Chapter 16, *File and Database Lookups.*

Domain, Host, and Address Lists

The list mechanism is the third facility that, together with string expansion and lookups, is the main building block of Exim configurations. Earlier, we showed the example:

```
local_domains = tiber.rivers.example:\
                *.cities.example:\
                dbm;/usr/exim/domains
```

in which the value of local_domains is a colon-separated list containing several types of patterns for matching a domain name. Similar list facilities are used for recognizing specific hosts and email addresses for particular purposes. The full description of lists is in Chapter 18, but we come across plenty of examples before then.

If a colon is actually needed in an item in a string list, it can be entered as two colons. Leading and trailing whitespace on each item in a string list is ignored. This makes it possible to include items that start with a colon, and in particular, certain forms of IPv6 address. For example:

```
local_interfaces = 127.0.0.1 : ::::1
```

defines the IPv4 address 127.0.0.1 followed by the IPv6 address ::1. Because the requirement to double colons is particularly unfortunate in the case of IPv6 addresses, a means of changing the separator was introduced with Exim Version 3.15.* If a list starts with a left-angle bracket followed by any punctuation character, that character becomes the list separator. The previous example could be rewritten as:

```
local_interfaces = <; 127.0.0.1 ; ::1
```

where the separator is changed to a semicolon.

The Default Qualification Domain

In a locally submitted message, if an unqualified address (that is, a local part without a domain) is found in the envelope or any of the header lines that contain addresses, it is qualified using the domain defined by qualify_domain (for senders) or qualify_recipient (for recipients) at the time the message is received. User agents normally use fully qualified addresses, but there are exceptions.

The default value for both these options is the name of the local host. If only qualify_domain is set, its value is used as a default for qualify_recipient. It is common in some installations to use these options to set a generic domain. For example, the Acme Widget Corporation might have two hosts handling its mail, *mail1.awc.example.com* and *mail2.awc.example.com*, but would probably require messages created on these hosts to use just *awc.example.com* as the default domain, rather than the individual hostnames. This can be done by setting the following:

```
qualify_domain = awc.example.com
```

in the Exim configurations on both hosts.

* This applies to all lists, with the exception of log_file_path andtls_verify_ciphers.

Handling Frozen Bounce Messages

When a message on Exim's queue is marked as *frozen*, queue runner processes skip over it and do not attempt to deliver it. One reason why a message might be frozen is mentioned in the section "Missing Data," in Chapter 3, *Exim Overview*; namely, there may be a problem with Exim's configuration. However, by far the most common reason that a message becomes frozen is that it is a bounce message that cannot be delivered. Such messages are often the result of incoming junk mail that is addressed to an unknown local user, but which contains an invalid sender address that causes the resulting bounce message to fail.*

In order to avoid mail loops, Exim does not let a failing bounce message give rise to another bounce message. Instead, Exim freezes the message to bring it to the postmaster's attention. On busy systems, frozen messages of this type may be quite common.

Some administrators do not have the human resources to inspect each frozen message in order to determine what the problem is, and their policy may be to discard such failures. Exim can be configured to do this by setting `ignore_errmsg_errors`, which has the effect of discarding failing addresses in bounce messages (the action is logged). A slightly less harsh option is to set `ignore_errmsg_errors_after`, which allows such failures to be kept for a given time before being discarded. For example, the following:

```
ignore_errmsg_errors_after = 12h
```

keeps such messages for 12 hours. After the first failure, the message is frozen as in the default case, but after it has been on the queue for the specified time, it is automatically unfrozen at the next queue run; if delivery fails again, the message is discarded.

Reducing Activity at High Load

In the main section of the configuration file, there are several options that allow you to limit or reduce Exim's activities when a lot of mail arrives at once, or when the system load is too high. "System load" in this sense is the average number of

* It is possible to do some checking on the sender and recipients before a message is accepted, as described in the section "Verifying Recipient Addresses," in Chapter 13, *Message Reception and Policy Controls*. This can dramatically cut down the number of frozen messages, but there may still be undeliverable messages that get through.

processes in the operating system's run queue over the last minute, a figure that can be obtained by running the *uptime* command to obtain output like this:

```
4:15pm  up 1 day(s), 22:23,  75 users,  load average: 0.09, 0.15, 0.22
```

The first of the "load average" figures is the one-minute average. On an unloaded system, it is a small number, usually well under 10. When it gets too high, everything slows down; reducing the load created by mail reception and delivery can alleviate the impact of this.

Delaying or Suspending Delivery When the Load Is High

By default, Exim starts a delivery process for each new message, and uses its queue for messages that cannot be delivered immediately. You can use various configuration options to modify Exim's behavior when system load is sufficiently high.

If the system load is higher than the value of `queue_only_load`, automatic delivery of incoming messages does not occur; instead, they wait on Exim's queue until the next queue runner process finds them. The effect of this is to serialize their delivery because a queue runner delivers just one message at a time. This reduces the number of simultaneously running Exim processes without significantly affecting mail delivery, as long as queue runners are started fairly frequently. For example, a setting of:

```
queue_only_load = 8
```

is a useful insurance against an overload caused by the simultaneous arrival of a large number of messages. If, on the other hand, `deliver_load_max` is set to:

```
deliver_load_max = 10
```

no deliveries at all are done if the load is higher that this setting, and if this is detected during a queue run, the remainder of the run is abandoned. A different threshold can be specified for queue runs by setting `deliver_queue_load_max`, for example:

```
deliver_queue_load_max = 14
```

If combined with the previous setting, this would allow deliveries only from queue runs when the load was between 10 and 14.

These three options are not fully independent. If `queue_only_load` (described earlier) is set, forcing all deliveries to take place in queue runs above a given load level, you can set either `deliver_load_max` or `deliver_queue_load_max` to a higher value in order to suspend all deliveries when the load is above that value. For example:

```
queue_only_load = 8
deliver_queue_load_max = 11
```

Setting both `deliver_load_max` and `deliver_queue_load_max` is useful only when `queue_only_load` is not set.

Deliveries that are forced with the *-M* or *-qf* command-line options override these load checks.

Suspending Incoming Mail When the Load Is High

There is no option for stopping incoming messages from local processes when the load is high, but mail from other hosts can be stopped or restricted to certain hosts. If `smtp_load_reserve` is set, and the system load exceeds its value, incoming SMTP calls over TCP/IP are accepted only from those hosts that match an entry in `smtp_reserve_hosts`. If this is unset, all calls from remote hosts are rejected with a temporary error code. For example, with the following:

```
smtp_load_reserve = 5
smtp_reserve_hosts = 192.168.24.0/24
```

only hosts in the 192.168.24.0/24 network can send mail to the local host when its load is greater than 5. The host list in `smtp_reserve_hosts` is also used by the `smtp_accept_reserve` option, which is described later.

If you are running user agents that submit messages by making TCP/IP calls to the local interface, you should probably add 127.0.0.1 (or ::1 in an IPv6 system) to `smtp_reserve_hosts`, to allow these submissions to proceed even at high load.

Controlling the Number of Incoming SMTP Connections

It's a good idea to set a limit on the number of simultaneous incoming SMTP calls, because each one uses the resources required for a separate process. Exim has the `smtp_accept_max` option for this purpose. The default setting is 20, which is reasonable for small to medium-sized systems, but if you are running a large system, increasing this to 100 or 200 is reasonable.

You can reserve some of these incoming SMTP slots for specific hosts. If you set `smtp_accept_reserve` to a value less than `smtp_accept_max`, that number of slots is reserved for the hosts listed in `smtp_reserve_hosts`. This feature is typically used to reserve slots for hosts on the local LAN so that they can never be all taken up by external connections. For example, if you set:

```
    smtp_accept_max = 200
    smtp_accept_reserve = 40
    smtp_reserve_hosts = 192.168.24.0/24
```

then once 160 connections are active, new connections are accepted only from hosts in the 192.168.24.0/24 network.

You can also set `smtp_accept_queue`; if the number of simultaneous incoming SMTP calls exceeds its value, automatic delivery of incoming SMTP messages is suspended; they are placed on the queue and left there for the next queue runner. The default for this option is unset, so that all messages are delivered immediately.

If new SMTP connections arrive while the daemon is busy setting up a process to handle a previous connection, they are held in a queue by the operating system, waiting for the daemon to request the next connection. The size of this queue is set by the `smtp_connect_backlog` option, which has a default value of 5. On large systems, this should be increased, say to 50 or more.

Checking for Free Disk Space

You can arrange for Exim to refuse incoming messages temporarily if the amount of free space in the disk partition that holds its spool directory falls below a given threshold. For example:

```
    check_spool_space = 50M
```

specifies that no mail can be received unless there is at least 50 MB of free space in which to store it.* The check is not a complete guarantee because of the possibility of several messages arriving simultaneously.

Limiting Message Sizes

It is a good idea to set a limit on the size of message your host will process. There is no default in Exim, but you can set, for example:

```
    message_size_limit = 20 M
```

to apply a limit of 20 MB per message.

Parallel Remote Delivery

If a message has a number of recipients on different remote hosts, Exim does these deliveries one at a time, unless you set `remote_max_parallel` to a value greater than one. On systems that are handling mostly personal mail, where messages typically have at most two or three recipients, this is not an important issue.

* Digits in a numerical option setting can always be followed by K or M, which cause multiplication by 1024 and 1024×1024, respectively.

However, on systems that are handling mailing lists, where a single address may end up being delivered to hundreds or even thousands of addresses, parallel delivery can make a noticeable improvement to performance. Setting, for example:

```
remote_max_parallel = 10
```

allows Exim to create up to 10 simultaneous processes to do remote deliveries for a message that has multiple recipients. Note that this option applies to the parallel delivery of individual messages; it is not an overall limit on Exim delivery processes.

Controlling the Number of Delivery Processes

In a conventional configuration, where Exim attempts to deliver each message as soon as it receives it, there is no control over the number of delivery processes that may be running simultaneously. On a host where processing mail is just one activity among many, this is not usually a problem. However, on a heavily loaded host that is entirely devoted to delivering mail, it may be desirable to have such control. It can be achieved by suppressing immediate delivery (which means that all deliveries take place in queue runs) and limiting the number of queue runner processes, for example, by placing these settings in the cofiguration file:

```
queue_only
queue_run_max = 15
```

Setting `queue_only` disables immediate delivery, and `queue_run_max` specifies the maximum number of simultaneously active queue runners. The maximum number of simultaneous delivery processes is then:

```
queue_run_max x remote_max_parallel
```

With this kind of configuration, you should arrange to start queue runner processes frequently (up to the maximum number) so as to minimize any delivery delay. This can be done by starting a daemon with an option such as *-q1m*, which starts a new queue runner every minute.*

Large Message Queues

Back in Chapter 3, we explained that Exim is designed for an environment in which most messages can be delivered almost instantaneously. Consequently, the queue of messages awaiting delivery is expected to be short. In some situations,

* See the section "The Daemon Process," in Chapter 11, *Shared Data and Exim Processes*, for more details of the daemon process.

nevertheless, large queues of messages occur, resulting in a large number of files in a single directory (usually called */var/spool/exim/input*). This can affect performance significantly. To reduce this degradation, you can set:

```
    split_spool_directory
```

When this is done, the input directory is split into 62 subdirectories, with names consisting of a single letter or digit, and incoming messages are distributed between them according to the sixth character of the message ID, which changes every second. This requires Exim to do a bit more work when it is scanning through the queue, but the directory access performance is much improved when there are many messages on the queue.

Large Installations

One of the advantages of Exim's decentralized design is that it scales fairly well, and can handle substantial numbers of mailboxes and messages on a single host. However, when the numbers start to get really large, a conventional configuration may not be able to cope. In this section, a number of general observations are made that are relevant to large installations.

Linear Password Files

Above a thousand or so users, the use of a linear password file is extremely inefficient, and can slow down local mail delivery substantially. Some operating systems (for example, FreeBSD) automatically make use of an indexed password file, or can be configured to do so, which is one easy way round this problem if you happen to be using such a system. The alternative is to make use of NIS or some other database for the password information, provided that it operates quickly.

Even if you don't have any login accounts on your mail server, you still need some kind of list of local users, and it is important to make the searching of this list as efficient as possible.

It is not only mail delivery that provokes password file lookups. If you are running a POP daemon, a password check happens every time a POP client connects; in environments where users remain connected and leave their POP MUAs running, these checks happen every few minutes for each user, whenever the POP client checks for the arrival of new mail.* IMAP is much less expensive than POP in this regard, because it establishes a session that remains active, so there is a password check only at the start.

* Users have been known to configure their MUAs to check as often as every 20 or 30 seconds; such usage will eat up your machine and should be strongly discouraged.

Mailbox Directories

You will get very bad performance if you have too many mailboxes in a single directory. What constitutes too many depends on your operating system; the default Linux filing system starts to degrade at about one thousand files in a single directory, whereas for Solaris the number is around ten thousand. This applies whether you are using individual files as multimessage mailboxes, or delivering messages as separate files in a directory.

The solution to this is to use multiple directory levels. For example, instead of storing *jimbo*'s mailbox in */var/mail/jimbo*, you could use */var/mail/j/jimbo*. Splitting on the initial character(s) of the local part is easy to implement, but it is not as good as using some kind of hashing function. Exim's string expansion facilities can be used to implement either a substring-based or hash-based split. Of course, you will have to ensure that all the programs that read the mailboxes use the same algorithm.

For a very large number of mailboxes, a two-level split is recommended, using Exim's numeric hash function, as in this example:

```
/var/mail/${nhash_8_512:$local_part}/$local_part
```

The hashing expansion generates two numbers separated by a slash, in this case using the local part as the data and ensuring that the numbers are in the ranges 0–7 and 0–511. This example places *jimbo*'s mailbox in */var/mail/6/71/jimbo*. The initial split could be between different disks or file servers, and the second one could be between directories on the same disk.

Simultaneous Message Deliveries

If two messages for the same mailbox arrive simultaneously, they cannot both be delivered at once if the mailbox is just a single file. One delivery process has to wait for the other, thus tying up resources. The default way that Exim does this (in the **appendfile** transport) is by sleeping for a bit, and then starting the process of locking the mailbox from scratch. This is the safest approach, and the only way to operate when lock files are in use.

Attempts to lock a mailbox continue for a limited time. If a process cannot gain access to a mailbox within that time, it defers delivery with the error message:

```
failed to lock mailbox
```

and Exim will try the delivery again later. If you see a lot of these messages in the main log file, it is an indication that there is a problem with contention for the mailbox.

If you are in an environment in which only `fcntl()` locks are used, and no separate lock files, you can configure the **appendfile** transport to use blocking calls, instead of sleeping and retrying. This gives better performance because a waiting process is released as soon as the lock is available instead of waiting out its sleep time. In this environment, this single change can make a big performance difference.

If the mailbox files are NFS-mounted, and more than one host can access them, you *must not* disable the use of lock files. If you do, you are likely to end up with mangled mailboxes.

The whole problem of locking can be bypassed if you use mailboxes in which each message is stored in a separate file.[*] One example of this type of message storage, called *maildir* format, is now quite popular, and has support in a number of MUAs and other programs that handle mailboxes. Because each message is entirely independent, no locking is required, several messages can be delivered simultaneously, and old messages can even be deleted while new ones are arriving. See the section "Maildir Format," in Chapter 9, *The Transports*, for a description of how to configure Exim to use maildir format.

Minimizing Name Server Delays

A busy general mail server makes a large number of calls to the DNS. For this reason, you should arrange for it to run its own name server, or make sure that there is a name server running on a nearby host with a high-speed connection, typically on the mail server's LAN. Ensure that the name server has plenty of memory so that it can build up a large cache of DNS data.

Storing Messages for Dial-up Hosts

You should not plan to store large numbers of messages for intermittently connected clients in Exim's spool. As explained in the section "Intermittently Connected Hosts," in Chapter 12, it is much better to have them delivered into local files, for onward transmission by some other means.

Hardware Configuration

If you keep increasing the workload of an Exim installation, disk I/O capacity is what runs out first. Each message that is handled requires the creation and deletion of at least four files. Large installations should therefore use disks with as high

[*] There is still some locking, of course, between processes that are updating the mailbox directory, but it is handled internally in the file system and is no longer Exim's responsibility.

a performance as possible. Also, it does not make sense to keep on increasing the performance of the processor if the disks cannot keep up.

Better overall performance can be obtained by splitting up the work between a number of different hosts, each with its own set of disks. For example, separate hosts can be used for incoming and outgoing mail. A general form of scalable configuration that is used by some very large installations is shown in Figure 4-2.

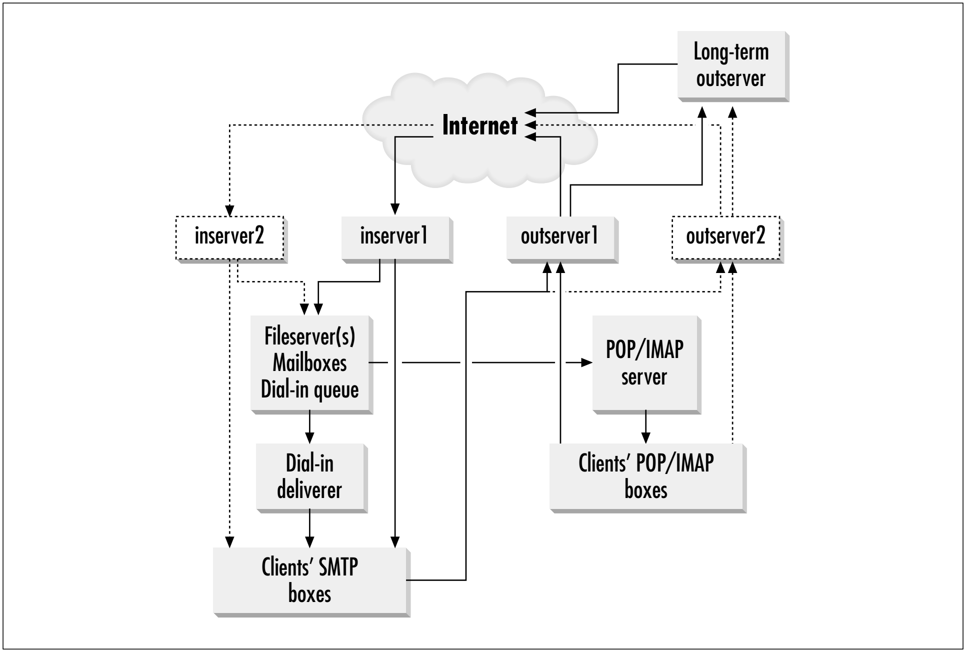


Figure 4-2. Large system configuration

This configuration has separate servers for incoming and outgoing messages, and can be expanded "sideways" by the addition of more servers (indicated by the dashed lines) as necessary. Incoming mail is delivered to one or more file servers, which hold local mailboxes in a split directory structure, as described earlier, and also messages that are waiting for dial-up hosts. The mailboxes are accesssed from POP and IMAP servers, and the dial-up hosts use yet another server to access their stored mail.

The outgoing servers send messages that they cannot deliver in a short time to a long-term outgoing server, so as not to impact their performance with very long message queues. This can be implemented using `fallback_hosts` on appropriate drivers on the main servers, or using the $message_age variable to move messages after some fixed time.

5

Extending the Delivery Configuration

In Chapter 3, *Exim Overview*, we describe the basics of how Exim delivers messages and works through a simple, straightforward example. The chapters that follow this one cover all the different drivers and their options, but before we descend into such detail, we'll look at some further examples of fairly common delivery requirements and discuss ways of configuring Exim to support them. In many cases, the suggested solution is not necessarily the only possible approach; there are often several ways of achieving the same result. The main intent of this chapter is to show you some more of the many ways in which the driver options can be used.

Multiple Local Domains

Any number of domains can be designated as local by listing them in `local_domains`. For example:

```
local_domains = simple.example : *.simple.example
```

If there are many local domains, it is cumbersome to include the list in the configuration file, and it is better to refer to a file instead. A setting such as:

```
local_domains = /etc/local.domains
```

could be used with a file containing lines such as:

```
simple.example
^[^.]*\d{4}\.simple\.example$
```

The second line is a regular expression that matches domains ending in *.simple.example* whose first component ends with four digits. The file can contain any type of item that may appear in a domain list, except for another filename. It is read each time it is needed, and so can be updated independently of Exim's configuration file. However, it is still scanned linearly, just like an in-memory list, and this can be slow when the number of items in the list is large. If the list contains only fixed names (that is, no wildcarded items of any kind), it can be converted into an indexed file that can be searched more quickly, by a setting such as:

```
local_domains = dbm;/etc/local.domains.db
```

This form of list entry specifies a lookup type (that is, a way of looking something up) as well as additional data required by the lookup, separated by a semicolon. In this example, the lookup type is `dbm` and the additional data is a filename.

There are several different software libraries that support indexed datafiles; *DBM* is a generic term that refers to this kind of file access method.* Most modern operating systems have a suitable library installed as standard. As a user of Exim, all you really need to know is that an indexed file gives quicker access to specific data, just as an index in a book allows you to find something more quickly than reading through. The details of how the index is implemented inside the DBM library are not important.

Although DBM libraries support the addition and removal of individual entries in a DBM file, the usual approach for applications that are just using the file as a fast way of accessing fixed data is to rebuild the file from scratch whenever the data changes. A utility program called *exim_dbmbuild* is supplied to do this job.

In order to determine whether a domain is local or not, Exim uses the domain name as the key for an indexed lookup. If data with a matching key is found in the file, the domain is local. The data that was looked up is not itself used in this case; Exim is interested only in whether or not the key exists in the file.

You do not have to put every local domain into a single lookup, because a lookup is just one item in the list that is searched. For example, you could have some domains inline, and some in one or more lookups:

```
local_domains = maindomain.example : dbm;/etc/otherdomains.db
```

Exim processes lists from left to right, so it makes sense to put the most commonly expected domains first.

DBM is only one of several lookup types supported by Exim; whenever a lookup is permitted, any of the available types may be used. For example, a list of local domains could be held in a MySQL database, and checked by a setting such as:

```
local_domains = mysql;select * from domainlist where domain='$key'
```

* DBM probably once stood for "database management," but nobody ever spells it out in full any more.

When processing a list of this sort, the variable `$key` contains the name of the domain that is being checked, so that it can be included in database queries such as this.

However, in the case of `local_domains`, using a database such as MySQL is not recommended, because `local_domains` is central to Exim's handling of addresses, so you want any lookups that are involved to be as fast as possible. The availability of the data for `local_domains` is also important. If you use (for example) an LDAP lookup as the only entry in `local_domains`, all mail delivery ceases if the LDAP server becomes unavailable. For these reasons, if the number of local domains is too large for an inline list, it is best to use an indexed file on a local disk.

Differentiating Between Multiple Domains

If you define a set of local domains using `local_domains`, and make no other changes to the configuration, Exim treats all of them as synonymous, with the same local part at any one of them being handled in the same way. In order to distinguish between different domains, the directors have to be made to act differently for different domains. The usual way this is done is to set the `domains` option on one or more directors. This option provides a list of domains for which the director is to operate. Here is a simple example with two local domains, each with its own alias file:

```
local_domains = a.local.domain : b.local.domain

a.aliases:
  driver = aliasfile
  domains = a.local.domain
  file = /etc/a.aliases
  search_type = lsearch

b.aliases:
  driver = aliasfile
  domains = b.local.domain
  file = /etc/b.aliases
  search_type = lsearch
```

Addresses of the form *user@a.local.domain* are processed by the first director, but not the second, whereas *user@b.local.domain* is processed by the second and not the first. If there are a number of domains whose alias filenames follow a regular pattern, there is no need to have a separate director for each one, because the file name can be varied depending on the domain, and a single director can be used:

```
aliases:
  driver = aliasfile
  file = /etc/$domain.aliases
  search_type = lsearch
```

In this example, all domains are processed, but because the filename contains the expansion variable `$domain`, a different file is used for each domain. Using the string expansion features of Exim, much more complicated transformations of the domain name are possible, including, for example, looking up a domain's alias filename from a file or database.

Virtual Domains

The term *virtual domain* is used to refer to a domain in which all the valid local parts are aliases for other addresses. There are no real mailboxes associated with a virtual domain. Because each address that is generated by an aliasing operation is independently processed, the result of handling an address in a virtual domain can be the address of a local mailbox, or a remote address that causes the message to be sent to another host.

The aliasing scheme just described can be used to handle virtual domains with a separate alias file for each domain. This makes it easy to have a separate maintainer for each file. However, it is important to consider what happens when a local part does not match any item in an alias file. Exim would normally offer the address to the next director. However, for a virtual domain, no further directors should be run, and instead the address should fail.

One way of doing this is for all subsequent directors to have a setting of `domains` that excludes the virtual domains, but a shortcut is provided by the `no_more` option, which specifies that no more directors are to be run after the current one. Here is an extract from a configuration file that handles a mixture of real and virtual domains:

```
local_domains = real.domain.example : cdb;/etc/virtuals

...

virtuals:
  driver = aliasfile
  domains = cdb;/etc/virtuals
  file = /etc/$domain.aliases
  search_type = lsearch
  no_more

system_aliases:
  driver = aliasfile
  file = /etc/aliases
  search_type = lsearch

...
```

The list of virtual domains in this example is kept in an indexed file in *cdb* format. This is a format that is optimized for files that are never updated after they have been created, and it performs better than conventional DBM, which allows for both reading and writing.*

The virtual domains are handled by the first director only; the real domain is handled by the remaining directors (of which only the first is shown here). The lookup in the */etc/virtuals* file happens twice in principle (once to establish that the domain is local, and again to check before running the first director), but Exim caches the results of the last lookup on a per-file basis, so the file is read only once in practice.

For more complicated requirements (for example, when some virtual domains are synonymous, and therefore use the same alias file), multiple directors can be used, or the name of the alias file for each domain can be looked up by a setting such as:

```
file = ${lookup{$domain}cdb{/etc/virtuals}{$value}}
```

This is an example of the use of a file lookup from within an expansion string. A description of the syntactic niceties is left until later, but you can see that it is triggered by the word `lookup`, and various substrings are provided, some within curly brackets (braces).

The key for the lookup is the content of `$domain`, the lookup type is `cdb`, and the indexed file is */etc/virtuals*. We know the lookup is going to succeed, because it is the same lookup that was used by `domains` to control the running of the director.† After performing the lookup, the final substring is expanded, with the data that was looked up contained in `$value`, so the result of the expansion in this case is just that string.

The data that is used to create the *cdb* file for this example could contain lines such as this, where the first two domains use the same alias file:

```
virt10.example:   /etc/virt1.aliases
virt11.example:   /etc/virt1.aliases
virt20.example:   /etc/virt2.aliases
```

Another approach to virtual domains that is sometimes used when the alias lists are not too large and are managed by a single person is to keep a single list for all of them that contains entries such as this:

```
jan@virt1.example:   J.Smith@dom1.example
jim@virt1.example:   J.Smith@dom2.example
jan@virt2.example:   J.Jones@dom1.example
jim@virt2.example:   J.Joyce@dom3.example
```

* See the section "Single-Key Lookup Types," in Chapter 16, *File and Database Lookups*, for more details.

† In fact, because there is lookup caching, the lookup isn't repeated; the cached result is reused.

Note that this differs from a "traditional" alias file in that the aliases are listed with domains attached, instead of just being local parts. The **aliasfile** director can use data like this, but only if you set the `include_domain` option. Without it, only the local part of the address is used when looking up aliases. The director might look like this:

```
virtuals:
  driver = aliasfile
  domains = cdb;/etc/virtuals
  file = /etc/virtual.aliases
  search_type = cdb
  include_domain
  no_more
```

There is scope for confusion if local parts without domains are used in alias files. Consider the following:

```
jac@virt2.example:   J.Hawkins
```

What domain should be added to *J.Hawkins* to make it a fully qualified address? Does it refer to a user on the local host, or should it retain the incoming domain *virt2.example*? Unless you tell it otherwise, Exim assumes that unqualified local parts are local, and it uses the value of `qualify_recipient` (a main configuration option) to create a complete address. If you want the other behavior, you must set `qualify_preserve_domain` on the **aliasfile** driver.

Defaults in Virtual Domains

When you configure a virtual domain, you may want to trap unknown local parts and forward them to a designated address such as the postmaster. Adding an asterisk to the search type causes Exim to look for a single asterisk entry if it cannot find the original address, so a configuration such as:

```
virtuals:
  driver = aliasfile
  domains = cdb;/etc/virtuals
  file = /etc/$domain.aliases
  search_type = lsearch*
  no_more
```

would allow a separate default to be specified for each domain in its alias file. For example, in the file */etc/virt3.example.aliases*, you could have:

```
*:          postmaster@virt3.example
postmaster: pat@dom5.example
jill:       jkr@dom4.example
```

When Exim looks up a local part other than `postmaster` or `jill` in this file, it fails. Because of the asterisk in the search type, it then does a second lookup for the key string `*`, which finds the default entry that directs all other local parts to the postmaster.

Putting the default entry first in a file that is linearly searched is a good idea, because it is then found quickly. This may look like a wildcard that would match any keystring, including *postmaster* and *jill*, but this is not the case. The asterisk is just being used as a special key that means "default."

If lookup defaulting is used when domains are included as part of lookup keys (that is, when `include_domain` is set), it provides a single default for all the domains. You can give each domain its own default by adding `*@` (instead of just `*`) to the search type. For example:

```
virtuals:
  driver = aliasfile
  domains = cdb;/etc/virtuals
  file = /etc/virtual.aliases
  search_type = lsearch*@
  include_domain
  no_more
```

Then you can include data such as:

```
*@virt4.example:  virt4-admin@dom6.example
```

in the combined virtual domain file. If an initial lookup fails, the local part is replaced with an asterisk for the second attempt, and only if that also fails is the plain asterisk tried as a key.

Postmasters in Virtual Domains

If each virtual domain has its own postmaster, these can be included in the alias data and there is no problem. Sometimes, however, there is a requirement for a single address to receive postmaster mail for a number of domains, and there is an easier approach than maintaining the same entry in all the alias files. If this applies to all local domains, a **smartuser** director, placed as the first director, can be used:

```
postmaster:
  driver = smartuser
  local_parts = postmaster
  new_address = postmaster@your.domain.example
```

The effect of setting `local_parts` is analagous to setting `domains` because it causes the director to be run only for those local parts that it matches. In this case, it runs only when the local part is *postmaster*, but as there is no `domains` setting, it runs for the postmaster address in any local domain.

The **smartuser** director itself applies no tests to the local part or domain. If it runs, it always succeeds. In this configuration, it is doing an aliasing operation, replacing any incoming address with the same fixed address. The new address is reprocessed in its own right; if *your.domain.example* is a local domain, it is processed again by this director, but on the third occasion, the antilooping rule takes effect, and the director is skipped. The unnecessary second pass through the director can be avoided in one of two ways:

- You can arrange to skip this director when the domain is already correct, by adding:

  ```
  domains = !your.domain.example
  ```

  to its configuration. An exclamation mark at the start of an item in the list negates the item, so this setting specifies that the director is to be run only when the domain is not *your.domain.example*.

- Alternatively, you can arrange to process the new address by starting at a specific director, instead of at the first one. This is done by setting `new_director` to the name of the director at which to start:

  ```
  postmaster:
    driver = smartuser
    local_parts = postmaster
    new_address = postmaster@your.domain.example
    new_director = system_aliases
  ```

  This goes straight to the **system_aliases** director.

If some of the virtual domains do have their own postmasters, but you want to pick up postmaster mail for the others, you can extend the `domains` setting to exclude the unwanted domains (or include the wanted ones, if that is easier). Another possibility, if defaults are not being used, is to place the **smartuser** director after those that handle the virtual domains.

Mailing Lists

Exim can be used on its own to run simple mailing lists that are maintained by hand, but for large or complicated requirements, the use of additional specialized mailing list software (such as Majordomo or SmartList) is strongly recommended.*

Lists of just a few addresses can be managed as aliases, but when larger lists are involved, it is usually more convenient to keep each list in a separate file. Also, this allows each list to be managed by its own manager, who need not have access to other Exim configuration files.

* For Majordomo, see *http://www.greatcircle.com/majordomo*, and for SmartList see *http://www.procmail.org* (sic).

The **forwardfile** director can be used to "explode" such mailing lists, and the `domains` option can be used if it is required to run these lists in a separate domain from normal mail. For example, if your domain is *simple.example*, you might want to use *lists.simple.example* for addresses that refer to mailing lists to keep them entirely separate from normal mail. This director does just that:

```
lists:
  driver = forwardfile
  domains = lists.simple.example
  no_more
  file = /usr/lists/$local_part
  no_check_local_user
  forbid_pipe
  forbid_file
  errors_to = $local_part-request@$domain
```

The domain must, of course, be set up as local, and appropriate MX records must be created in the DNS if it is to be accessible from other hosts. If you use list names that are distinct from any of your local usernames, you can have them in your normal domain, and the `domains` and `no_more` settings are not needed.

The `no_check_local_user` option stops **forwardfile** from checking that the local part is the login ID of a local user, which it does by default (because it is most commonly used for users' *.forward* files). No check is made on the ownership of the file containing the list, because neither `owners` nor `owngroups` is set. The `forbid_pipe` and `forbid_file` options prevent a local part from being expanded into a filename or a pipe delivery, which is not normally appropriate for a mailing list.

The `errors_to` option specifies that any delivery errors caused by addresses taken from a mailing list are to be sent to the given address rather than the original sender of the message. In other words, it changes the envelope sender of the message as it passes through. However, before acting on `errors_to`, Exim verifies the error address. If verification fails, the envelope sender is not changed. In the earlier example, verification succeeds if the *-request* file that corresponds to the mailing list exists.

Using this scheme, you can create a list by creating the main file containing the members, and the *-request* file containing the managers. For example, as soon as a file called */usr/lists/exim-users* is created, mail to *exim-users@lists.simple.example* is accepted, and sent to all the addresses in that file. As soon as */usr/lists/exim-users-request* is created, verification of the address *exim-users-request@lists.simple.example* succeeds, allowing the envelope sender address of messages to *exim-users@lists.simple.example* to be changed when they are forwarded.

An alternative to handling both the list address and the errors address with the same director is to set up an earlier director to handle the errors address. An example of this is shown in the section "Closed Mailing Lists," later in this chapter.

Syntax Errors in Mailing Lists

If an entry in a forward file contains a syntax error, Exim normally defers all deliveries for the original address. This may not be appropriate when the list is being maintained automatically from address texts supplied by users, because a single bad address shuts down the entire list.

If `skip_syntax_errors` is set on the **forwardfile** director, the director just skips entries that fail to parse, noting the incident in the log. Valid addresses are recognized and used. If in addition `syntax_errors_to` is set to a verifiable address, messages about skipped addresses are sent to it. It is usually approriate to set this to the same value as `errors_to`.

NFS-Mounted Mailing Lists

It is not advisable to have list files that are NFS-mounted, because the absence of the mount cannot be distinguished from a nonexistent file. Thus, Exim would behave as if a list did not exist when the NFS server was down. One way around this problem is to use an **aliasfile** director with the alias file containing a list of lists that are kept on local disk. This makes the existence or nonexistence of a list clear.

Each alias expansion can then be an "include" item to read the list itself from a separate, NFS-mounted file. If `no_freeze_missing_include` is set for the **aliasfile** director, an unavailable file causes delivery to be deferred without freezing. For example, the director could be:

```
lists:
  driver = aliasfile
  file = /usr/list.of.lists
  search_type = lsearch
  no_freeze_missing_include
  forbid_pipe
  forbid_file
  errors_to = $local_part-request@$domain
```

with the alias file containing lines such as:

```
exim-users:   :include:/usr/lists/exim-users
```

This is a bit more complicated to maintain, because in addition to creating the files, the aliases have to be updated in order to set up a new list.

Reexpansion of Mailing Lists

In order to avoid duplicate deliveries, Exim remembers every individual address to which a message has been delivered, but it normally stores only the original recipient addresses with a message. If all the deliveries to a mailing list cannot be done at the first attempt, the mailing list is reexpanded when the delivery is next tried. This means that alterations to the list are taken into account at each delivery attempt, and as a consequence, addresses that have been added to the list since the message arrived will receive a copy of the message, even though it predates their subscription.

If this behavior is felt to be undesirable, the `one_time` option can be set on the **forwardfile** director. When this is done, any addresses generated by the director that fail to deliver at the first attempt are added to the message as "top level" addresses, and the address that generated them is marked "delivered." As a result, expansion of the mailing list does not happen again at subsequent delivery attempts. The disadvantage of this is that if any of the failing addresses are incorrect, changing them in the file has no effect on pre-existing messages.

Closed Mailing Lists

The examples so far have assumed open mailing lists, to which anybody may send mail. It is also possible to set up closed lists, where mail is accepted from specified senders only. This is done by making use of the `senders` option that restricts the running of a director or router to messages that have specific senders.

The following example uses the same file for each list, both as a list of recipients and as a list of permitted senders, but different or multiple sender lists could, of course, be used. For instance, a list for announcements could restrict senders to those people who are permitted to make the announcements.

First, it is necessary to set up a separate director to handle the `-request` address, to which anybody may send mail:

```
lists_request:
  driver = forwardfile
  domains = lists.simple.example
  suffix = -request
  file = /usr/lists/$local_part-request
  no_check_local_user
  no_more
```

Here we see a new option, `suffix`, which we have not met before. It has the effect of testing the local part for the given suffix, and bypassing the director if it does not match. This director, therefore, is run only for local parts that end with `-request`.

The director runs with `$local_part` stripped of the suffix, which is placed in $local_part_suffix (though that variable is not used in this example). There is an analagous option called `prefix`, which operates by testing the other end of the local part. You would use this if your mailing lists used the form *owner-xxx* for list management instead of *xxx-request*.

The next director handles the closed list itself:

```
lists:
  driver = forwardfile
  domains = lists.simple.example
  require_files = /usr/lists/$local_part
  senders = lsearch;/usr/lists/$local_part
  file = /usr/lists/$local_part
  no_check_local_user
  forbid_pipe
  forbid_file
  one_time
  skip_syntax_errors
  errors_to = $local_part-request@lists.simple.example
  no_more
```

The `require_files` option tests for the existence of one or more files, before running the director. It is needed here to ensure that the file exists before trying to search it using the `senders` option, because an attempt to search a nonexistent file causes an error. If the file does not exist (that is, if the mailing list is unknown), the director declines, but because `no_more` is set, no further directors are tried. Therefore, Exim fails the address.

If the file exists and contains the address of the sender, the director is run and the message is delivered to the list. However, if the file does not contain the sender, the director is not run, and no further directors are run because of `no_more`. Note that `senders` behaves differently from `domains` in the way it interacts with `no_more`, as explained in the section "Interaction of Conditions," in Chapter 6, *Options Common to Directors and Routers*.

External Mailing List Software

The use of specialized mailing list software such as Majordomo is recommended if you are running mailing lists in a big way. Messages addressed to a mailing list are handed off to an external program, which ultimately resubmits them to Exim for delivery to the subscribers. This can be done either by providing a recipient list with each resubmitted message, or by using Exim's aliasing or forwarding mechanisms to pick up lists of addresses from files. This approach helps you to deal with the following issues:

- Although Exim is capable of delivering a single message to thousands of recipients, having one large forwarding list is not the best way of handling a mailing list with thousands of subscribers. Remember that Exim routes or directs every address in a message before it does any deliveries. It does this serially, so if there are very many recipients, a long time may elapse after the arrival of a message before any deliveries are actually done. To avoid this, mailing list software should normally be configured to send multiple copies of messages, with a maximum of around one hundred recipients in each copy. This introduces some parallelism into the routing process. It is an advantage if the addresses can be sorted, to keep all those in the same domain together.*
- An external program makes it easier to check and possibily modify the contents of messages posted to your lists. For example, some lists prohibit the use of attachments; others require modification of header lines or the addition of standard texts to messages.
- The common practice of sending an email message for automatic subscription and unsubscription from lists can be supported only by using an external mailing list agent.

Different mailing list software packages provide different facilities. For example, automatic subscription might be supported without the ability to generate multiple copies of the message when there are large numbers of recipients. You should investigate several packages to see which one best fits your needs.

When external mailing list software is in use, Exim has to recognize certain local parts and pipe the messages to appropriate programs. Occasionally, there is also a requirement to recognize messages that have come back from the mailing list software, and process their addresses in some special way. Exim is usually configured to run as a specific mailing list user when delivering incoming messages through a pipe. For example, if you are using Majordomo and keeping all the mailing list information in a single alias file, you could use this director:

```
majordomo_aliases:
  driver = aliasfile
  file = /usr/local/majordomo/lists/majordomo.aliases
  search_type = lsearch
  user = majordom
  group = majordom
```

This ensures that any pipes that are run as a result of that particular alias file do so as the user *majordom*.

* Many general mailing lists contain lots of subscribers from big ISP domains such as *aol.com*. Unless you are using VERP (see the section "Changing the Return Path," in Chapter 9, *The Transports*), making sure the addresses are sorted minimizes the number of copies sent to such domains.

When an ordinary user submits a message to Exim from a process running on the same host, an envelope sender address is created from the user's login name and the domain in `qualify_domain` (which defaults to the hostname). There is a command-line option `-f` that can override this, but Exim ignores it unless the caller is *trusted*. Trusted users are discussed more fully in the section "Privileged Users," in Chapter 19, *Miscellany*, but the basic idea is that such users are allowed to forge sender addresses and other message data. If you are using Majordomo, for example, you should have the following:

```
trusted_users = majordom
```

in your Exim configuration, so that the `-f` option is honored for messages coming from Majordomo, thus allowing it to specify an appropriate envelope sender for each mailing list.

Using aliases as the means of directing messages to list management software is not the only possibility. Another approach is to use specialized directors and transports. For example, the following transport could be used to pipe messages to SmartList:

```
list_transport:
  driver = pipe
  command = /usr/slist/.bin/flist $local_part$local_part_suffix
  current_directory = /usr/slist
  home_directory = /usr/slist
  user = slist
  group = slist
```

The transport is activated from a director like this:

```
list_director:
  driver = smartuser
  suffix = -request
  suffix_optional
  local_parts = !.bin:!.etc
  require_files = /usr/slist/$local_part/rc.init
  transport = list_transport
```

The `require_files` option ensures that the director runs only when the local part is the name of an existing list. That is, the existence of a file whose name contains the list name is used as the trigger for passing the message to SmartList. Note the use of the `local_parts` option to avoid treating */usr/slist/.bin* and */usr/slist/.etc* as mailing lists.

Using an External Local Delivery Agent

An alternative to using the **appendfile** transport for writing to local mailboxes is to use an external program for this purpose. This could be for all local deliveries, or only for certain local parts. The **pipe** transport can be used to pass messages to a separate local delivery agent such as *procmail*.[*] We use *procmail* as an example of a local delivery agent in what follows, but a similar approach could be used for any local delivery agent.

Individual users can arrange for their mail to be delivered using *procmail* by calling it from their *.forward* files, provided that the Exim configuration permits the use of pipes from *.forward* files. In some installations, however, there may be a requirement always to use *procmail* for local deliveries, or to allow users to choose to use it without letting them run pipe commands from their *.forward* files. One way to handle these cases is to set up a separate transport just for the use of *procmail*.

When doing this, care must be taken to ensure that the pipe is run under an appropriate uid and gid. In some configurations, one wants this to be a uid that is trusted by the delivery agent to supply the correct sender of the message. It may be necessary to recompile or reconfigure the delivery agent so that it trusts an appropriate user. The following is an example of a transport that delivers using *procmail*:

```
procmail_pipe:
  driver = pipe
  command = /usr/local/bin/procmail -d $local_part
  return_path_add
  delivery_date_add
  envelope_to_add
  check_string = "From "
  escape_string = ">From "
  user = $local_part
  group = mail
```

This runs *procmail* with the user's uid, but with the group set to `mail`.

[*] See *http://www.procmail.org*. Many, but not all, of the things *procmail* can do can also be done using an Exim filter. See Chapter 10, *Message Filtering*, for a discussion of the differences.

The command specified for the transport does *not* begin with the following:

```
IFS=" "
```

as shown in some *procmail* documentation. This setting arose on systems where the MTA runs pipe commands via a shell; it ensures that the separator character between the arguments of a shell command is a space. Exim does not by default use a shell to run pipe commands, so if this shell construct is present, it is not recognized. Instead of using a shell, Exim itself splits up the command into separate arguments before it does string expansion. This means that any shell metacharacters that occur in substituted values (for example, in `$local_part`) cannot affect the parsing of the command. Exim then runs the command directly. This approach not only avoids problems with shell metacharacters, but also saves the overhead of starting another process.

The transport shown earlier could be used by a director, which checks that the user has a *.procmailrc* file:

```
procmail:
  driver = localuser
  transport = procmail_pipe
  require_files = .procmailrc
```

If there is no *.procmailrc* file in the user's home directory, this director declines to handle the address. The following director could be a conventional **localuser** director that directs to an **appendfile** transport in the usual way. Thus, all a user needs to do to change from Exim's normal delivery to delivery via *procmail* is to create *.procmailrc*. No *.forward* file is required.

The next example shows a transport for a system where local deliveries are handled by the Cyrus IMAP server:

```
local_delivery_cyrus:
  driver = pipe
  command = /usr/cyrus/bin/deliver \
            -m ${substr_1:${local_part_suffix}} \
            -- ${local_part}
  user = cyrus
  group = mail
  return_output
  log_output
  prefix =
  suffix =
```

Note the unsetting of `prefix` and `suffix`, and the use of `return_output` to cause any text written by Cyrus to be returned to the sender. This transport could be activated by a director such as this:

```
local_user_cyrus:
  driver = localuser
  transport = local_delivery_cyrus
```

Multiple User Addresses

A single user normally has a single email address and a single mailbox. For example, the user *caesar* on the host *simple.example* has the following address:

```
caesar@simple.example
```

and mail to that address is commonly delivered into */var/mail/caesar*. Users with high volumes of incoming mail often like to use some method of automatically sorting it into categories to make it more convenient to handle. One way of doing this is to make use of Exim's filtering capability or to run an external local delivery agent such as *procmail*. These methods rely on analyzing the header lines or message content.

Another approach is to allow the use of prefixes or suffixes on usernames in the local parts of incoming mail. For example, additional addresses such as the following:

```
caesar-rome@simple.example
casear-gaul@simple.example
```

are recognized as belonging to the user *caesar*. The user can then make use of forwarding or filtering files to inspect the suffix. Fixed suffixes could be specified, but usually the wildcard facility is used so that users can choose their own suffixes. For example, the director shown in the following:

```
userforward:
  driver = forwardfile
  file = .forward
  suffix = -*
  suffix_optional
  filter
```

runs a user's *.forward* file (usually this would be an Exim filter file) for all local parts that start with a valid username, optionally followed by a hyphen and then arbitrary text. Within a filter file, the user can distinguish different cases by testing the variable `$local_part_suffix`. For example:

```
if $local_part_suffix contains -special then
  save /home/$local_part/Mail/special
endif
```

If the filter file does not exist or does not deal with such addresses, they are offered to subsequent directors, and assuming no subsequent use of the `suffix` option is made, those with suffixes presumably fail. Thus, users have control over which suffixes are valid.

An alternative way of differentiating between suffixes in local parts is to arrange for a suffix to trigger the use of a different *.forward* file. This has the advantage that the user does not need to learn about Exim filter files. For example:

```
userforward:
  driver = forwardfile
  file = .forward${local_part_suffix}
  suffix = -*
  suffix_optional
  filter
```

If there is no suffix, *.forward* is used; if the suffix is `-special`, for example, *.forward-special* is used. Once again, if the appropriate file does not exist, or does not deal with the address, it is offered to subsequent directors. The user controls which suffixes are valid by creating appropriate files, which may forward messages to other addresses or direct them to specific files or pipe commands using traditional *.forward* features or Exim filter commands.

Mixed Local/Remote Domains

Consider a corporate mail gateway that delivers some local parts in one particular domain into local mailboxes, and sends others on to personal workstations. The domain is local in the Exim sense, but the deliveries to workstations are remote deliveries. To implement this, a mapping from local parts to workstation names is required; for example, the following:

```
ceo:      bigcheese.plc.co.example
alice:    castor.plc.co.example
bob:      pollux.plc.co.example
```

means that mail for the local part *ceo* is to be sent on to the host *bigcheese.plc.co.example*, and so on. The first director might be as follows:

```
workstation:
  driver = smartuser
  local_parts = lsearch;/etc/wsusers
  transport = local_smtp
```

Local parts that are not present in the file cause the director to be skipped, and they can then be processed by subsequent directors as conventional local users, whereas any local part that is found in the file is sent to the **local_smtp** transport, which could be configured thus:

```
local_smtp:
  driver = smtp
  hosts = $local_part_data
```

Earlier examples of the **smtp** transport have not used the `hosts` option, because they have been referenced from routers that supply a list of hosts for delivery. In this case, however, the transport is referenced from a director, which cannot pass a host list, so the list must appear on the transport itself. The variable $local_part_data contains the data from the lookup in the `local_parts` director option. So, in this case, if the local part is *ceo*, $local_part_data contains *bigcheese.plc.example*, the host to which the message is to be sent.

Another approach to the same situation is not to define the domain as a local domain, thereby causing its addresses to be offered first to the routers. The first router picks off the local parts that are to be delivered to workstations:

```
workstation_people:
  driver = domainlist
  domains = plc.co.example
  local_parts = lsearch;/etc/wsusers
  route_list = * $local_part_data byname
  transport = remote_smtp
```

We haven't come across the **domainlist** router before. It is the main router used for manually routing certain domains; that is, for implementing rules of the form "send mail for this domain to that host." Normally all such routers are placed early in the list, followed by a final **lookuphost** router to deal with those domains that are not special in any way.

In this example, we've restricted this router to the *plc.co.example* domain and to those local parts that are found in the */etc/wsusers* file. The routing rule is specified by the `route_list` option, which has three parts:

- The asterisk means "for all domains," but because of the setting of `domains`, we know the domain is actually *plc.co.example*. Another way of configuring this router would be to omit the `domains` setting, and replace the asterisk by *plc.co.example*. This would be slightly less efficient because it would not do the domain check until Exim was actually running the router.
- The second part of the rule is the name of the host to which the message should be sent. The value of $local_part_data is the result of the lookup that was done to match the local part, which in this case is exactly the workstation name we need.

- The word `byname` at the end specifies how the IP address of the host is to be looked up, in this case by calling the system function `gethostbyname()`. Most operating systems allow this to be configured to search */etc/hosts* and possibly other data sources, including the DNS.*

Because the router is defining the host to which the message is to be sent, the standard **remote_smtp** transport can be used.

So far, we've dealt with the local parts that get sent on to workstations. What about those that are to be delivered locally? They will bypass the first **domainlist** router because they will not be found in */etc/wsusers*. What we want to happen is for them to get passed to the directors. This can be done by setting up a second special router:

```
local_people:
  driver = domainlist
  domains = plc.co.example
  route_list = * localhost byname
  self = local
```

This routes the domain *plc.co.example* to the local host. The default action on discovering that an apparently remote domain routes to the local host is to freeze the message, because actually sending it in the normal way would probably create a tight loop. However, for special cases like this, the `self` option can be used to tell Exim to do something different. In this case, the setting of `self` specifies that anything routed to the local host by this router is to be treated as a local domain. Any address in the *plc.co.example* domain that reaches this router is therefore passed to the directors. This is a useful trick that can often be used to advantage for domains that need to be partly routed and partly directed.

Delivering to UUCP

Exim contains no special UUCP features, and in particular, it does not support UUCP's "bang path" method of addressing. However, if you stick to using Internet domain addresses, mail can easily be routed to UUCP. First of all, you need to set up a mapping from domain names to UUCP hostnames. This could be a file containing data such as:

```
darksite.plc.example:  darksite
bluesite.plc.example:  indigo
```

* The file */etc/nsswitch.conf* is often used to specify how a system lookups up hostnames.

Then you need a router that uses this data:

```
uucphost:
  transport = uucp
  driver = domainlist
  route_file = /usr/local/uucpdomains
  search_type = lsearch
```

This router searches the file */usr/local/uucpdomains*. If it finds the domain, it places the data it found in the `$host` variable, and routes the address to the **uucp** transport:

```
uucp:
  driver = pipe
  user = nobody
  command = /usr/local/bin/uux -r - $host!rmail $local_part
  return_fail_output = true
```

Because this is a local transport, the entries in the routing file must contain just a single hostname (as they do in this example). Using this configuration, a message addressed to the following:

```
postmaster@darksite.plc.example
```

would end up being piped to the command:

```
/usr/local/bin/uux -r - darksite!rmail postmaster
```

which is run as the user *nobody*.

Ignoring the Local Part in Local Deliveries

Local deliveries are not required to make use of the local part of an address. One common example is a small company that has only one person reading incoming email. Rather than set up fixed local parts such as *sales*, *info*, *enquiries*, and so on, they want all mail delivered into a single mailbox, whatever the local part. One way of doing this would be to set up a default alias, as described in the section "Defaults in Virtual Domains," earlier in this chapter, but it does not even have to be this complicated. Assuming the chosen recipient is *postmaster*, all you need is the following director:

```
catchall:
  driver = smartuser
  new_address = postmaster
```

It should be placed last in the list of directors, so that it picks up all unknown local parts and redirects them to *postmaster*.

Some ISPs allocate a domain name to each small account, and then deliver all messages addressed to that domain into a single mailbox, ignoring local parts. For Exim running on such an ISP's mail server, there are two issues to consider:

- How to find the mailbox from the domain name
- How to enable the owner of the mailbox to distinguish between different recipients

Suppose that the ISP allocates the domain name *diego.isp.example* to a customer whose username is *diego* and whose mailbox is */var/mail/diego* on the ISP's mail host to which the lowest-numbered MX record points. The Exim configuration on the mail host could set the following:

```
local_domains = *.isp.example
```

to recognize all such domains as local; a single **smartuser** director can pick up these addresses and direct them to a special transport:

```
onebox_customers:
  driver = smartuser
  domains = *.isp.example
  transport = onebox
```

The transport extracts the mailbox name from the domain and delivers to the relevant file:

```
onebox:
  driver = appendfile
  file = /var/mail/${if match{$domain}{^([^.]+)}{$1}}
  user = ${if match{$domain}{^([^.]+)}{$1}}
  envelope_to_add
  return_path_add
```

The value the `file` option introduces a new kind of expansion item, starting with `$if{`. In general, `if` tests a condition, and expands a substring only if the condition is met.* In this example, the condition is that the contents of `$domain` match a regular expression. The regular expression `^([^.]+)` matches a string that begins with a sequence of non-dot characters, and saves that sequence in `$1`. Since the domain has already been checked, we know that in this case the regular expression will always match. For our example, the string expansion:

```
${if match{$domain}{^([^.]+)}{$1}}
```

causes the regular expression to be applied to the string created by substituting the domain name, that is, to *diego.isp.example*. It matches and saves the substring `diego` in the variable `$1`. This is then substituted in the final set of braces, giving `diego` as the result of the whole `if` expansion item.

* See Chapter 17, *String Expansion*, for a full explanation.

The same expansion is used to specify the user that is used to run the delivery. Setting `envelope_to_add` and `return_path_add` has the effect of preserving the envelope addresses in header lines, so it is possible for the owner of the mailbox to distinguish between messages to different local parts, even if the recipient addresses do not appear in the *To:* header lines.

Handling Local Parts in a Case-Sensitive Manner

RFC 822 states that the case of letters in the local parts of addresses must be assumed to be significant. (In contrast, the case of letters in domain names is never significant.) Exim preserves the case of local parts in remote addresses, in accordance with the RFC. However, on most Unix systems, usernames are in lowercase, and local parts in email addresses are expected to be handled without regard to case, so that messages addressed to:

```
icarus@knossos.example
Icarus@knossos.example
ICARUS@knossos.example
iCaRuS@knossos.example
```

are all delivered to the local user *icarus*. By default, therefore, whenever it is processing an address in one of its local domains, Exim forces the local part into lowercase. This behavior can be disabled by setting:

```
locally_caseless = false
```

If you do this, the four addresses in the previous example would be treated as having different local parts on the host where *knossos.example* is a local domain. However, sites that use mixed-case usernames do not usually have accounts that differ only in the case of their letters. They generally still want to have case-insensitive treatment of local parts in email. That is, they still want to recognize local parts without regard to the case of their letters, but deliver them to case-sensitive mailboxes. Unsetting `locally_caseless` is therefore not sufficient; you also have to arrange to convert local parts to the correct casing. One way of doing this is to set up the first director as a **smartuser** director to do the conversion by a file lookup such as:

```
adjust_casing:
  driver = smartuser
  new_address = ${lookup{${lc:$local_part}}cdb\
                {/etc/usercased.cdb}{$value}fail}\
                @$domain
```

This director generates a new address by using the `lc` expansion item to force the local part into lowercase, and then looking up the lowercased version in (in this example) a *cdb* file, whose data might contain entries such as:

```
icarus:   Icarus
j.caesar: J.Caesar
```

Thus, all four addresses previously listed would be turned into *Icarus@knossos.example*. The new address has the correct casing, and can therefore be successfully looked up in the password file by other directors.

For maximum efficiency, a director such as this should also contain a setting of the `new_director` option. Without it, the new address is processed afresh, and if it is different from the original address, it passes through the **adjust_casing** director for a second time, though this just regenerates the same new address. The next time around, the director is skipped because it has already processed that address. Setting `new_director` to the next director, as shown in the following example:

```
new_director = system_aliases
```

avoids the wasted second pass through **adjust_casing**.

Scanning Messages for Viruses

There are a number of programs that will scan an email message to determine whether it contains any viruses as attachments. Some of these run on Unix systems, but others are available only for other operating systems. The general technique for using such a program from Exim is the same in both cases: incoming messages are delivered to the scanner, which has a secure means of passing those that are "clean" back to Exim for final delivery. This process causes an additional *Received:* header to be added to the message, but that is not usually a problem.

Messages can be diverted to the scanner program directly from a director or router, or a system filter can be used in some cases. We have not discussed Exim's filtering facilities yet (details are in Chapter 10), but in brief, a system filter allows a message to be inspected before it is delivered, and various actions can be taken, depending on what the filter finds. In the context of virus scanning, this can save some resources by doing some initial testing before calling an external scanner. However, as discussed here, using a system filter introduces some complications that can be avoided if a filter is not used.

As an example of how a filter might be used, consider checking for attachments. MIME messages that have attachments must contain a header line such as:

```
Content-Type: multipart/mixed;
  boundary="------------9D6D28528332819A908698F9"
```

A "boundary" has to be defined for there to be any attachments. You could test for this in a system filter by this command:

```
if $h_Content-Type: contains "boundary" then ...
```

and pass the message to an external scanner only if the condition was met.

Attachments are not the only way that viruses can be transmitted. In practice, you would need some additional tests, such as checking for the absence of uuencoded material, before skipping the full virus check.

Virus Checking on the Local Host

If your virus scanner runs on the local host, Exim can deliver a message to it via a pipe, and it can be returned by running a new Exim reception process and passing the message back through another pipe. It is important to preserve the original sender address. This can be done by making use of the `-f` command-line option, or by transferring the message in batch SMTP format. In both cases, the process that returns the message to Exim must be running as a trusted user, because only trusted users are permitted to specify sender addresses.

The normal way of identifying messages that have come back from the scanner is to make use of the `-oMr` command-line option. This defines the protocol by which the message is received. Normally, it is set to values such as `smtp` or `local`, but trusted callers can set it to an arbitrary string. The value is recorded in the log and is available in the `$received_protocol` variable, but it is not otherwise used by Exim.

In order to make this work, therefore, you need to decide which uid is to be used to run the scanner, and to set it up as an Exim trusted user. It is a good idea to reserve a special uid just for this purpose. Suppose you have set up a user called *vircheck*. Adding the following:

```
trusted_users = vircheck
```

makes it trusted. This means that any process running under the uid of *vircheck* can supply arbitrary sender addresses and protocol values for messages it submits.

Next, you need to set up a transport to pipe the message to the scanner. The following example uses batch SMTP (BSMTP):

```
pipe_to_scanner:
  driver = pipe
  command = /path/to/scanner/command
  user = vircheck
  bsmtp = all
  prefix =
```

Setting `bsmtp` to `all` means that only a single copy of the message is sent, however many recipients there are. The setting of `prefix` to an empty string is important; the default is to insert a UUCP-like `From` line, which is not correct for BSMTP format.

Using BSMTP makes it easy for the scanner to return the message with the sender and recipients unchanged. If the special protocol value is `scanned-ok`, for instance, the scanner can run a command of the form:

```
exim -oMr scanned-ok -bS
```

and copy the message it received to its standard input, without modification. For example, the following Perl script is a dummy scanner that does not actually do any checking, but simply resubmits the message:

```
#!/usr/bin/perl
open(OUT, "|exim -oMr scanned-ok -bS")
  || die "Failed to set up Exim process\n";
print OUT while (<STDIN>);
close(OUT);
```

You now have to arrange for new messages to be passed to the scanner. For addresses in local domains, you can do this by inserting a new director at the start of the list of directors:

```
send_to_scanner:
  driver = smartuser
  transport = pipe_to_scanner
  condition = ${if eq {$received_protocol}{scanned-ok}{no}{yes}}
```

The `condition` option checks the value of `$received_protocol` and skips the director if its value is `scanned-ok`. Without this check, messages would loop forever between Exim and the scanner. For messages not received from the scanner, all local addresses are directed to the **pipe_to_scanner** transport, which batches them into a single delivery to the scanner program because of the setting of `bsmtp`.

If you also want messages addressed to remote domains to be scanned, you need to create a new router as well:

```
send_to_scanner:
  driver = domainlist
  transport = pipe_to_scanner
  condition = ${if eq {$received_protocol}{scanned-ok}{no}{yes}}
  route_list = *
```

This should be placed first in the routers configuration, and could be confined to certain domains by replacing the asterisk in `route_list` with a suitable pattern.

An alternative way of arranging for messages to be passed to a scanner program is to make use of a system filter, which will apply whether the recipients are local or remote. However, in several ways this is more complicated. The filter would contain a command such as this:*

```
if $received_protocol is not scanned-ok then
  pipe "/the/scanner/command $sender_address '$recipients'"
endif
```

When a system filter sets up a delivery in this way, it is considered to have handled the delivery arrangements for the message, so normal delivery to the regular recipient addresses is bypassed, and the only delivery that is done is to the scanner. In this example, the sender address is passed to the command as the first argument, and the list of recipients (which are separated by a comma and a space) is the second argument, created by expanding `$recipients`.† To return a clean message for delivery, the scanner should call Exim with the equivalent of this command line:

```
exim -oi -oMr scanned-ok -f '$sender_address' '$recipients'
```

and write the message to the standard input. Although no additional director or router is required, you still have to define a transport. It should not have a `command` setting, because the command is specified by the filter. So all that is needed is:

```
pipe_to_scanner:
  driver = pipe
  user = vircheck
  prefix =
```

but in addition you must set:

```
message_filter_pipe_transport = pipe_to_scanner
```

to tell Exim which transport to run when it encounters a pipe command in a system filter. Batch SMTP cannot be used in this case, because this is a special kind of one-off delivery for the whole message, independent of the normal recipients.

There is one other complication that arises when a system filter is used like this to create an extra delivery. The message's recipients are added to the message in an

* See Chapter 10 for a discussion of filter commands.

† The `$recipients` variable is available only in system filters. For privacy reasons, it is not available in user filters.

X-Envelope-To: header line. It is not normally appropriate to leave this line in the message when it is resubmitted, so it must be recognized and removed. It could be used as another way of obtaining the list of recipients in the scanner.

Virus Checking on an External Host

If your virus checker runs on an external host, all that is needed is a configuration that sends messages to the checking host, unless they arrived from there. Suppose the checker is running on IP address 192.168.13.13; a transport to deliver messages there is:

```
smtp_to_scanner:
  driver = smtp
  hosts = 192.168.13.13
```

The director to send unchecked messages to the scanner is now:

```
send_to_scanner:
  driver = smartuser
  transport = smtp_to_scanner
  condition = ${if eq {$sender_host_address}{192.168.13.13}{no}{yes}}
```

and the equivalent router for remote addresses is:

```
send_to_scanner:
  driver = domainlist
  transport = smtp_to_scanner
  condition = ${if eq {$sender_host_address}{192.168.13.13}{no}{yes}}
  route_list = *
```

If you have more than one host running a virus checker, you can specify the transport as:

```
smtp_to_scanner:
  driver = smtp
  hosts = 192.168.13.13 : 192.168.14.14 : ...
  hosts_randomize
```

The `hosts_randomize` option makes Exim put the hosts in a random order before trying them, each time the transport is run. The condition in the director or router is now more complicated:

```
send_to_scanner:
  driver = smartuser
  transport = smtp_to_scanner
  condition = ${if or {\
                {eq {$sender_host_address}{192.168.13.13}}\
                {eq {$sender_host_address}{192.168.14.14}}\
                ...
                }{no}{yes}}
```

It is not straightforward to use a system filter directly when an external checker is in use because of the complication of preserving the message's recipients. A system filter could, however, add a header line that is used by directors and routers to determine whether to send a message to the checker or not.

Modifying Message Bodies

There have been a lot of questions on the Exim mailing list about modifying the bodies of messages as they pass through the MTA. There are three specific things that people want to do:

- Lawyers want to add standard disclaimers.
- Marketroids want to add advertisements.
- The paranoid want to remove attachments.

There are legal and ethical issues that arise in this connection that will not be discussed here, but there are also serious technical problems. Since the advent of MIME (RFC 2025), message bodies are no longer (in general) just strings of text characters. Just adding extra characters on the end of a message is likely to break the syntax of the message, causing it to become unreadable by standard MUAs.* Furthermore, if the message is digitally signed, making any change to the body invalidates the signature. Digital signatures are becoming more widely used now that they are legally binding in some countries.

Finally, it is not the job of an MTA to modify the bodies of messages. If such modification is to be undertaken, the program that does it needs a lot of specialist knowledge about the format of messages, which is inappropriate to include in an MTA.

If, despite all these considerations, you find yourself wanting to do this kind of thing using Exim, you need to make use of a transport filter, as described in Chapter 9. A transport filter lets you pass outgoing messages through a program or script of your choice. It is the job of this script to make any changes to the message that you require. By this means, you have full control over what changes are made, and Exim does not need to know anything about message bodies. However, using a transport filter requires additional resources, and may slow down mail delivery.

You can use Exim's directors and routers to arrange for those messages that you want to modify to be delivered via a transport filter. For example, suppose you want to do this for messages from addresses in your domain that are being

* One such client (now obsolete) crashes on encountering a message that has been tampered with in this way.

delivered to a remote host. You would place the following router before the standard **lookuphost** router:

```
filter_remote:
  driver = lookuphost
  transport = remote_smtp_filter
  condition = ${if eq {$sender_address_domain}{your.domain}{yes}{no}}
```

We've previously encountered the options `domains`, `local_parts`, `senders`, and `require_files`, which apply conditions to the running of routers and directors. These exist independently because those are commonly required tests. To cope with less common requirements, the `condition` option exists. Its value is a string that is expanded. If the result is `0`, `no`, or `false`, the driver is bypassed. For any other values, the driver is run (assuming other conditions are met, of course). This facility allows customized conditions to be applied, using any of the features available in string expansions.

This particular expansion tests contents of the variable $sender_address_domain, which contains the domain of the sender of the message. The router is run only if the value is *your.domain*. The actual routing is using the DNS as normal, but if it succeeds, a special transport called **remote_smtp_filter** is used. When the sender's domain is not *your.domain*, addresses fall through to the normal router and are routed to the standard **remote_smtp** transport.

Another way to do this would be to use a single router, with an expanded string for the transport setting:

```
lookuphost:
  driver = lookuphost
  transport = ${if eq {$sender_address_domain}{your.domain}\
             {remote_smtp_filter}{remote_smtp}}
```

The new transport is defined thus:

```
remote_smtp_filter:
  driver = smtp
  transport_filter = /your/filter/command
```

The entire message is passed to your filter command on its standard input. It must write the modified version to the standard output, taking care not to break the RFC 822 syntax. As this is a remote transport, the command is run as the Exim user.

6

Options Common to Directors and Routers

Previous chapters introduced a number of general options that can be set for any director or router to show how some common configuration requirements could be met. This chapter recaps those options, and also includes the remaining options that are common to all directors and routers. These are usually called *generic* options. In addition, each director and router has some options that are specific to its operation; these are described in the following chapters in the sections on the individual drivers.

One generic option that is always set is `driver`. This determines which particular director or router is to be used. When a driver decides to accept an address and queue it for a transport, the value of the generic `transport` option is expanded, and must yield the name of an available transport. If it does not, delivery is deferred.

The remaining generic options can be divided into four types:

- Those that set conditions for the running of the driver
- Those that change what happens when a driver is successful; that is, when it accepts an address
- Those that add data to an address accepted by the driver, for use when that address is delivered
- Those that provide debugging information

The options for any driver may be given in any order in the configuration file, except that those specific to the individual driver must follow the setting of `driver`. For this reason, `driver` is normally given first.

Conditional Running of Routers and Directors

In simple configurations, an address is offered to all the defined directors or routers (as appropriate) in turn, until one is found that can handle it, or until all have been tried. As shown in several examples in the previous chapter, one way of extending a simple configuration to deal with more complicated situations is to apply conditions to some directors (or routers) so that they are not run for every address, but only when the conditions are met.

Restricting Drivers to Specific Domains

The `domains` option is used to specify that a particular driver is to run only when the address contains certain domains, as in the virtual domains example from the previous chapter:

```
virtuals:
  driver = aliasfile
  domains = cdb;/etc/virtuals
  file = /etc/$domain.aliases
  search_type = lsearch
  no_more
```

Stopping an Address from Being Passed On

The virtual domains example also shows the use of `no_more`—an option that prevents subsequent drivers from running when one declines. This makes it easier to restrict one or more drivers to a given set of domains.

Restricting Drivers to Specific Local Parts

Just as the `domains` option restricts a driver to specific domains, the `local_parts` option restricts a driver to specific local parts. An example of this is when directing postmaster mail for a group of local domains to the same address:

```
postmaster:
  driver = smartuser
  local_parts = postmaster
  new_address = postmaster@your.domain.example
```

Restricting Drivers to Specific Senders

The `senders` option restricts a driver to messages with certain senders only. Earlier, we saw an example of how it could be used to implement a closed mailing list.

Restricting Drivers by Other Conditions

The `domains`, `local_parts`, and `senders` options exist because these are common conditions that are often required. For those less common, there is an option called `condition` whose value is an arbitrary string. This is run through the string expander, and if the result is a forced failure, an empty string, or one of the strings `0`, `no`, or `false`, the driver is not run. There are some examples of this in the discussion of virus scanning in the previous chapter. Because of the flexibility of the string expansion mechanism, a wide range of conditions can be tested.

Suppose you are prepared to deliver small messages directly over the Internet, but want to send large ones to some other host (which might, for example, send them out overnight). The standard **lookuphost** router could be modified like this:

```
lookuphost:
  driver = lookuphost
  transport = remote_smtp
  condition = ${if < {$message_size}{500K}{yes}{no}}
```

The expansion string is using a numeric comparison (specified by a "less than" sign) on the variable `$message_size`, which contains the size of the message. If the message is less than 500 KB in size, the string expands to `yes`; otherwise it expands to `no`. Thus, this router handles messages that are smaller than 500 KB only. You would, of course, also need a subsequent router to deal with the large messages. A **domainlist** router could be used for this.

Restricting a Driver to Verification Only

Exim draws a distinction between processing an address in order to deliver a message to it, and *verifying* an address to check that it is valid. Verification is mostly used to check the validity of addresses during incoming SMTP calls, at the point when the message's envelope is received. It consists of running the address though the directors or routers, as if it were being processed for delivery. An address verifies successfully if one of the directors or routers accepts it. Verification is discussed in detail in Chapter 13, *Message Reception and Policy Controls*.

Sometimes, particularly in the case of the directors, you want drivers to behave differently when verifying an address, as opposed to processing it for delivery. For example, while verifying that a local part refers to a user mailbox, there is no point wasting time checking the user's *.forward* file. Setting the following:

```
no_verify
```

on a director or router causes it to be skipped during verification. In fact, this is just a shorthand for:

```
no_verify_sender
no_verify_recipient
```

which suppress the driver during verification of senders and recipients independently.

As an example of where this could be useful, suppose you are using a **smartuser** director to pass all unrecognized local parts to a script that tries to generate helpful error messages, or to a different host that might be able to handle these addresses. This means that no local part that is passed to the directors will ever cause a failure during message delivery. However, if verification of senders at SMTP time is configured, you do not want arbitrary local parts in your domain to be accepted as valid incoming senders. The solution is to set `no_verify_sender` on the **smartuser** director, so that it is not run when verifying senders.

There may be occasional circumstances where it is helpful to run a director or router only during verification and not during delivery. This can be configured by setting:

```
verify_only
```

in its configuration. If you really want to, by making use of `verify_only` and `no_verify`, you can partition the routers or directors into a set that is used only for delivery, and another set that is used only for verification.

Restricting Drivers by File Existence

The final common conditional option for directors and routers is `require_files`. It causes the existence (or nonexistence) of certain files to determine whether a driver is run, and there are several examples of its use in the previous chapter. Here is the director that was shown as an example of how to call SmartList:

```
list_director:
  driver = smartuser
  suffix = -request
  suffix_optional
  local_parts = !.bin:!.etc
  require_files = /usr/slist/$local_part/rc.init
  transport = list_transport
```

The string value of `require_files` is expanded and then interpreted as a colon-separated list of absolute pathnames. If any string in the list is empty, it is ignored. Otherwise, except as described here, each string must be a fully qualified file path, optionally preceded by an exclamation mark (indicating negation).

A failure to expand the string, or the presence of a path within it that is not absolute, causes Exim to write a message to its panic log and exit immediately. This includes forced failure, because the whole string is expanded once, before being interpreted as a list. If you want a particular variant of the expansion to specify that no files are to be checked, you should cause it to yield an empty string rather than forcing failure.

If the option is used on a **localuser** director, or on a **forwardfile** director that has either of the `check_local_user` or `file_directory` options set, the expansion variable `$home` may appear in the list, referring to the home directory of the user whose name is that of the local part of the address.

The driver is skipped if any required path does not exist, or if any path preceded by `!` does exist. If Exim cannot determine whether or not a file exists, delivery of the message is deferred. This can happen when NFS-mounted filesystems are unavailable.

Sometimes an attempt to check a file's existence yields the error "Permission denied." This means that the user is not permitted to read one of the directories on the file's path. The default action is to consider this a configuration error, and delivery is deferred because neither the existence nor the nonexistence of the file can be determined. However, in some circumstances it may be desirable to treat this condition as if the file did not exist. If the filename (or the exclamation mark that precedes the filename for nonexistence) is preceded by a plus sign, this error is treated as if the file did not exist. For example:

```
    require_files = +/some/file
```

These file checks are normally run under the Exim uid. However, when processing an address for delivery, it is possible to arrange for this test to be run under a specific uid and gid. This may sometimes be a better way of avoiding the "Permission denied" problem. If an item in a `require_files` list does not contain any slash characters, it is taken to be the user (and an optional group, separated by a comma) to be used for testing subsequent files in the list. If no group is specified, but the user is specified symbolically, the gid associated with the uid is used; otherwise the gid is not changed. For example:

```
    require_files = mail:/some/file
    require_files = $local_part:$home/.procmailrc
```

The second example works because the `require_files` string is expanded before it is used. However, it would need to be on a director such as **localuser** that sets the `$home` variable appropriately.

When processing an address for verification (as opposed to delivery), you cannot cause the uid and gid to be changed in this way. This is because Exim has normally given up its privilege when accepting incoming SMTP mail. Often, it is possible to set `no_verify` on drivers that make use of the facility. Otherwise, some other approach to the problem must be found.

Interaction of Conditions

This section describes how the various conditional options described in the previous sections interact with each other.

If the domain and local part of an address are not in agreement with `domains` and `local_parts`, if the `condition` option fails, or if `verify_only` is set and verification is not happening, the director or router is skipped and the next one is tried.

Otherwise, if `no_more` is set, no subsequent drivers are ever called automatically.* The current driver is itself called except in these three circumstances:

- Verification is happening and its `verify_sender` or `verify_recipient` option (as appropriate) is turned off.
- The existence or nonexistence of files listed in the `require_files` option is not as expected.
- The sender of the message is not in agreement with `senders`.

The options `domains`, `local_parts`, `require_files`, `senders`, and `condition` are expanded and tested in that order. When any test fails, no further expansions are done. The order of testing can make a difference if some of them contain expansion items that refer to values relevant to the others. For example, suppose some director has these settings:

```
domains = dom1.example : dom2.example
local_parts = cdb;/etc/$domain
```

When `local_parts` is tested, $domain can only be *dom1.example* or *dom2.example*, and as long as the corresponding files exist, the test of the local part can be done. If `local_parts` were tested first, the domain would not be constrained, and might cause a lookup error as a result of a nonexistent file.

Because the `senders` and `condition` tests are done after checking for file existence, they can contain references to files whose existence is tested (for example, by looking up something in them).

* They can, however, be called when a router explicitly passes an address to the following driver, for example, by means of the `self` option or the `host_find_failed` option of the `domainlist` router.

Changing a Driver's Successful Outcome

When a router or director successfully handles an address, that address is normally considered to be finished with, and it is not passed to any more routers or directors. Suppose, however, that you want to save copies of messages addressed to certain recipients.* Using a router or director to set up deliveries of the required copies is a convenient way to handle this requirement, but of course the messages must go on to be delivered normally as well. The `unseen` option is provided for just this purpose. It is the complement of `no_more`; `unseen` causes directing or routing to continue when it would otherwise cease. Thus, the director below sends a copy of every message addressed to *ceo@plc.example* to *secretary@plc.example*, without disturbing the normal delivery:

```
copy_ceo:
  driver = smartuser
  local_parts = ceo
  domains = plc.example
  new_address = secretary@plc.example
  unseen
```

For this to work, *plc.example* must be defined as a local domain, and this director must precede the director that sets up the normal delivery. The `domains` setting is necessary only if the directors are handling different local domains in different ways.

Sometimes, when verifying addresses, you may want to force a specific address to fail. If a driver has `fail_verify` set and succeeds in handling an address that is being verified, verification fails instead of succeeding. Actually, `fail_verify` is just a shorthand for:

```
fail_verify_sender
fail_verify_recipient
```

which control this forced failure mechanism independently for sender and recipient addresses. These options make it possible to fail verification for a set of local parts that is defined by what a specific director matches. They apply only during verification, and are ignored when processing an address for delivery. Here is an example of a real case where this is used.

When an account is cancelled on one of the central systems at the University of Cambridge, it is not immediately removed from the password data; instead the password is reset and the home directory is set to */home/CANCELLED*. This makes it easy to reinstate an account. Only after some time has passed is it completely removed. Mail for these cancelled accounts must not be accepted, and therefore

* If you want to save copies of *all* messages independently of the recipients, it is best to use a message filter (see Chapter 10, *Message Filtering*) because it can arrange for a single copy, however many recipients the message has.

verification of their addresses must fail, so that incoming SMTP mail for them is rejected. Action is needed to achieve this, because the usernames are still in the password data.

A feature of the **localuser** director that we haven't yet met helps to solve this problem. It is called `match_directory`, and it causes the director to decline unless the home directory read from the password data matches the option's value. The simplest thing to do would be to add:

```
match_directory = /home/$local_part
```

to the normal **localuser** directory, so that accounts that do not have a normal home directory are not recognized. This achieves two things:

- Incoming SMTP messages for cancelled users are rejected because the recipient address fails to verify.
- If a message addressed to a cancelled user is received from a local process (where recipient verification does not happen), it is bounced because directing the address fails.

The problem with this is that the error given in bounce messages is "unknown local part," which often causes the senders to pester the postmaster with questions as to what has happened to the account "that worked yesterday." Therefore, we use a different strategy. The following additional director is inserted before those that handle local users:

```
cancelled_users:
  driver = localuser
  match_directory = /home/CANCELLED
  transport = cancelled_user_pipe
  fail_verify
```

This director checks for a local user whose home directory is */home/CANCELLED*; other local parts are passed on to the next director. When delivering, the message is passed to a **pipe** transport in order to generate a bounce message that explains what has happened (see the section "The pipe Transport," in Chapter 9, *The Transports*, for details of **pipe** transports). Setting `fail_verify` ensures that when addresses are being verified, any that are matched cause a verification error.

Adding Data for Use by Transports

When a director or router accepts an address, it can attach data to it for use when the address is finally transported. Some items are relevant only when the driver passes the address directly to the transport; others accumulate as the address passes through several drivers (for example, multiple instances of aliasing or forwarding).

Adding or Removing Header Lines

The header section of a message can be modified by adding or removing individual header lines at the time it is transported. Such modifications naturally apply only to the copy of the message that the transport writes out. This is not a very common requirement, but some installations find it useful.

Header line additions and removals can be specified on directors and routers (as well as on transports) by setting `headers_add` and `headers_remove`, respectively. Note, however, that addresses with different `headers_add` or `headers_remove` settings cannot be transported as multiple envelope recipients on a single copy of the message. This may reduce performance.

Each of these options sets a string that is expanded at directing or routing time, and retained for use at transport time. If the expansion is forced to fail, the option has no effect. For `headers_remove`, the expanded string must consist of a colon-separated list of header names, not including their terminating colons. For example:

```
headers_remove = return-receipt-to:acknowledge-to
```

For `headers_add`, the expanded string must be in the form of one or more RFC 822 header lines, separated by newlines (coded as `\n` inside a quoted string). For example:

```
headers_add = "X-added-header: added by $primary_hostname\n\
               X-another: added at time $tod_full"
```

Exim does not check the syntax of these added header lines, except that a newline is supplied at the end if one is not present. If an address passes through several directors or routers as a result of aliasing or forwarding operations, any `headers_add` or `headers_remove` specifications are all retained for use by the transport. This makes it possible to add header lines that record aliasing and forwarding operations on the address. For example, you could add the following:

```
headers_add = X-Delivered-To: $local_part@$domain
```

to the configuration for every aliasing and forwarding director. A message addressed to *postmaster@example.com* that was aliased to *p.master@example.com* and then forwarded to *pat@example.com* would end up with these added headers:

```
X-Delivered-To: postmaster@example.com
X-Delivered-To: p.master@example.com
```

This provides a complete record of the way the address was handled.

Because the addition of header lines does not actually happen until the message is transported, such lines are not accessible to subsequent directors or routers that may handle an address after it passed through a driver with `headers_add` set.

At transport time, removal applies only to the original header lines that arrived with the message, plus any that were added by a system filter. It is not possible to remove header lines that are added by a director or router. For each address, all the original header lines listed in `headers_remove` are removed, and those specified by `headers_add` are added in the order in which they were attached to the address. Then any additional header lines specified by the transport are added.

If you are making use of the `unseen` option to take copies of messages, header lines that are added by drivers that have `unseen` set are not present in the other deliveries of the message.

Changing the Return Path

When an address undergoes aliasing or forwarding, it is sometimes desirable to change the address to which subsequent bounce messages will be sent. This is variously known as the *return path*, *error address*, or *envelope sender*.

The most common case is when a mailing list is being "exploded." Bounce messages should return to the manager of the list, not the poster of the message. The generic option `errors_to` can be used on any director or router to change the envelope sender for deliveries of any addresses it handles or generates. If the address subsequently passes through other directors or routers that have their own `errors_to` settings, these override any earlier settings. The value of the option is expanded, and checked for validity by verifying it. It is not used if verification fails. The director that we used for mailing lists in the previous chapter is the following:

```
lists:
  driver = forwardfile
  domains = lists.simple.example
  no_more
  file = /etc/lists/$local_part
  no_check_local_user
  forbid_pipe
  forbid_file
  errors_to = $local_part-request@lists.simple.example
```

It shows a very typical use of `errors_to`.

Controlling the Environment for Local Deliveries

If a director or router queues an address for a local transport, the `user` and `group` options can be used to specify the uid and gid under which to run the local delivery process. One common case where the `user` option is needed is when an alias file sets up a delivery to a pipe or a file. For example, if */etc/aliases* contains this line:

```
majordomo:  |/usr/mail/majordomo ...
```

then either the **aliasfile** director or the transport to which it sends such deliveries must define the uid and gid under which the pipe command is to be run. The director's configuration could contain, for example:

```
user = majordom
```

The `user` and `group` options are strings that are expanded at the time the director or router is run, and must yield either a digit string or a name that can be looked up from the system's password data.* By this means, different values can be specified for different circumstances.

Suppose you are running another program via a pipe from an alias file, in addition to Majordomo. Perhaps it is an automated way of obtaining some kind of help, so that your aliases are:

```
majordomo:  |/usr/mail/majordomo ...
autohelp:   |/usr/etc/autohelp ...
```

Each pipe must run under its own uid, so a fixed value such as the one shown here is no longer possible. An expanded string can be used to select the correct user like this:

```
user = ${if eq {$local_part}{majordomo}{majordom}{autohelp}}
```

This expansion string checks the local part for the value `majordomo` and expands to `majordom` if it matches; otherwise it expands to `autohelp`.

If `user` is given without `group`, the group associated with the user is used as a default. For most directors and routers, the default for these options is unset, but for the **forwardfile** director with `check_local_user` set, and for the **localuser** director, the default is taken from the password data.

The uid and gid set by a director or router can be overridden by options on the transport. Another way to handle this example is to put the setting of `user` on the transport instead of the **aliasfile** director. If you do this, however, you would probably want to set up a dedicated transport for all the pipe commands generated by this director, so that other pipe commands (for example, from users' *.forward* files) do not use a transport with this `user` setting. You can use `address_pipe_transport` on the **aliasfile** director to specify which transport is used for the pipe commands generated within that director.

* Exim uses the operating system's functions for looking up users and groups. These may consult */etc/passwd* and */etc/group*, but in larger systems the password data is usually kept elsewhere, such as in NIS or NIS+ databases.

A user may be a member of many groups, but (at least on some operating systems) it is quite an expensive operation to find them and set them all up when changing a process's uid, so Exim does not do this by default. If you want this to be done, then in addition to setting `user` you need to set `initgroups`, which is a Boolean option that takes no data. For example:

```
user = majordom
initgroups
```

Specifying fallback_hosts

The final option that sets up data to be passed to a transport is `fallback_hosts`. This option provides a "use a smart host only if delivery fails" facility. It applies only to remote transports, and its value must be a colon-separated list of hostnames or IP addresses. It is not string expanded. If the transport is unable to deliver to any of the normal hosts, and the errors are not permanent rejections, the addresses are put on a separate transport queue with their host lists replaced by the fallback hosts.

For example, you might want to set up a host to attempt remote deliveries according to the normal MX routing, and to send any messages that cannot immediately be delivered to a smart host. You would use a standard **lookuphost** router to do the MX processing:

```
lookuphost:
  driver = lookuphost
  transport = remote_smtp
```

The associated transport makes use of `fallback_hosts`:

```
remote_smtp:
  driver = smtp
  fallback_hosts = smart.host.example
```

When a message is being delivered, Exim first tries the hosts that are found by the router from the MX records. If all of them give temporary errors, the address, instead of being deferred, is put onto a queue of addresses that are awaiting fallback processing.

Once normal delivery attempts are complete, the fallback queue is processed by rerunning the same transports with the new host lists. It is done this way (instead of trying the fallback hosts as soon as the ordinary hosts fail) so that if several failing addresses have the same `fallback_hosts` (and `max_rcpt` permits it), a single copy of the message, with multiple recipients, is sent.

There is one situation in which fallback hosts are not used. For any address that is routed using MX records, if the current host is in the MX list (that is, it is an MX backup for the address), fallback hosts are not used for that address for the following reason. Suppose a host is using a configuration such as the one previously shown, and is a secondary MX for some domain. When the primary MX host for that domain is down, mail for the domain arrives at the secondary host. It cannot deliver it to the primary MX host (because it is down), but it must not send it to its fallback host, because that host is likely to send it straight back, causing a mail loop.

Debugging Directors and Routers

There is an option called `debug_print` whose sole purpose is to help debug Exim configurations. When Exim is run with debugging turned on,* the value of `debug_print` is expanded and added to the debugging output that is written to the standard error stream (*stderr*). This happens before checking `require_files` and `condition`, but after the other conditional checks. This facility can be used to check that the values of certain variables are what you think they should be.

For example, if a `condition` option appears not to be working, `debug_print` can be used to output the values it references. In the section "Scanning Messages for Viruses," in the previous chapter, we use this router for sending messages to a virus scanner program:

```
send_to_scanner:
  driver = domainlist
  transport = pipe_to_scanner
  condition = ${if eq {$received_protocol}{scanned-ok}{no}{yes}}
  route_list = *
```

This router should be skipped if `$received_protocol` has the value `scanned-ok`. Suppose that this configuration was not working, and you were trying to find out why. One obvious approach is to check the value of `$received_protocol` at the time the router is run. You can do this by adding the following:

```
debug_print = received_protocol=$received_protocol
```

to the router, running a delivery with debugging turned on, and examining the standard error output.

* See *-d* and *-v* in the section "Options for Debugging," in Chapter 20, *Command-Line Interface to Exim.*

Summary of Director/Router Generic Options

The generic options that are applicable to both directors and routers are summarized in this section:

`condition` (string, default = unset)
: This option specifies a test that has to succeed for the driver to be called. The string is expanded. If the result is a forced failure, an empty string, or one of the strings `0`, `no`, or `false` (checked without regard to the case of the letters), the driver is not run.

`debug_print` (string, default = unset)
: If this option is set and debugging is enabled, the string is expanded and included in the debugging output.

`domains` (domains list, default = unset)
: This option restricts a director or router to specific mail domains. The string is expanded and interpreted as a colon-separated list. Because of the expansion, any items containing backslash or dollar characters must be escaped with a backslash. The driver is skipped unless the current domain matches an item in the list. If the match is achieved by means of a file lookup, the data that the lookup returned for the domain is placed in the $domain_data variable for use in string expansions of the driver's private options, and in the options of any transport that the driver sets up.

`driver` (string, default = unset)
: This option must always be set. It specifies which of the available directors or routers is to be used.

`errors_to` (string, default = unset)
: Delivery errors for any addresses handled or generated by the director or router are sent to the address that results from expanding this string, provided that it verifies as a valid address.

`fail_verify` (Boolean, default = false)
: Setting this option has the effect of setting both `fail_verify_sender` and `fail_verify_recipient` to the specified value.

`fail_verify_recipient` (Boolean, default = false)
: If this option is true when the driver successfully verifies a recipient address, verification fails instead of succeeding. This option has no effect if the `verify_recipient` option is false.

`fail_verify_sender` (Boolean, default = false)

If this option is true when the driver successfully verifies a sender address, verification fails instead of succeeding. This option has no effect if the `verify_sender` option is false.

`fallback_hosts` (string list, default = unset)

If a driver queues an address for a remote transport, this host list is associated with the address and used instead of the transport's fallback host list. String expansion is not applied to this option. The argument must be a colon-separated list of hostnames or IP addresses.

`group` (string, default = see description)

If a driver queues an address for a local transport, and the transport does not specify a group, the group given here is used when running the delivery process. For most directors and routers the default is unset, but for the **forwardfile** director with `check_local_user` set, and for the **localuser** director, the default is taken from the password data.

`headers_add` (string, default = unset)

This option specifies a string of text that is expanded at directing or routing time, and associated with any addresses that are processed by the driver. If the expanded string is empty, or if the expansion is forced to fail, the option has no effect. Other expansion failures are treated as configuration errors.

`headers_remove` (string, default = unset)

The string is expanded at directing or routing time and is then associated with any addresses that are processed by the driver. If the expansion is forced to fail, the option has no effect. Other expansion failures are treated as configuration errors. After expansion, the string must consist of a colon-separated list of header names.

`initgroups` (Boolean, default = false)

If the driver queues an address for a local transport, and this option is true, and the uid supplied by the router or director is not overridden by the transport, the `initgroups()` function is called when running the transport to ensure that any additional groups associated with the uid are added to the secondary groups list.

`local_parts` (string list, default = unset)

This option restricts a director or router to specific local parts. The string is expanded and interpreted as a colon-separated list. Because of the expansion, any items containing backslash or dollar characters must be escaped with a backslash. The driver is run only if the local part of the address matches the list. If the match is achieved by a lookup, the data that the lookup returned for the local part is placed in the variable `$local_part_data` for use in expansions of the driver's private options, and in the options of any transport that the driver sets up.

`more` (Boolean, default = true)
If this option is false and the driver runs but declines to handle an address, no further drivers are tried, and directing or routing fails. If the only reason a driver does not run is because `no_verify` is set (which can happen when an address is being verified, and all other conditions are met), a false value for `more` again prevents any further drivers from running. However, in all cases, if a router explicitly passes an address to the following router by means of the setting:

```
self = pass
```

or otherwise, the setting of `more` is ignored.

`require_files` (string list, default = unset)
This option checks for the existence or nonexistence of specified files or directories. Its value is expanded and interpreted as a colon-separated list of strings. If the option is used on a **localuser** director, or on a **forwardfile** director that has either of the `check_local_user` or `file_directory` options set, the expansion variable `$home` may appear in the list, referring to the home directory of the user whose name is that of the local part of the address. If any string is empty, it is ignored. Otherwise, each string must be a fully qualified file path, optionally preceded by `!` (indicating negation). The driver is skipped if any paths not preceded by `!` do not exist, or if any paths preceded by `!` do exist.

`senders` (address list, default = unset)
This option restricts a director or router to messages that have specific sender addresses. The string is expanded and interpreted as a colon-separated list of senders, in the same format as used for general options such as `sender_reject`.

`transport` (string, default = unset)
Some directors and routers require a transport to be supplied, except when `verify_only` is set, where it is not relevant. Others require that a transport not be supplied, and for some it is optional. The string must be the name of a configured transport after expansion. This allows transports to be dynamically selected.

`unseen` (Boolean, default = false)
Setting this option has a similar effect to the `unseen` command qualifier in filter files. It causes a copy of the incoming address to be passed on to subsequent drivers, even when the current one succeeds in handling it.

`user` (string, default = see description)
: If the driver queues an address for a local transport, and the transport does not specify a user, the user given here is used when running the delivery process. For most directors and routers, the default for `user` is unset, but for the **forwardfile** director with `check_local_user` set, and for the **localuser** director, the default is taken from the password data.

`verify` (Boolean, default = true)
: Setting this option has the effect of setting both `verify_sender` and `verify_recipient` to the specified value.

`verify_only` (Boolean, default = false)
: If this option is set, the driver is used only when verifying an address or testing with the `-bv` option, not when actually doing a delivery, testing with the `-bt` option, or running the SMTP `EXPN` command. The driver can be further restricted to verifying only senders or recipients by means of `verify_sender` and `verify_recipient`.

`verify_recipient` (Boolean, default = true)
: If this option is false, this driver is skipped when verifying recipient addresses.

`verify_sender` (Boolean, default = true)
: If this option is false, this driver is skipped when verifying sender addresses.

7

The Directors

We've met all four of Exim's directors in earlier chapters. Here is a list to remind you of them:

aliasfile
A director that expands aliases into one or more different addresses.

forwardfile
A director that handles users' *.forward* files and Exim filter files.

localuser
A director that recognizes local usernames.

smartuser
A director that accepts any address; it is used as a "catch-all."

In this chapter, there are separate sections for each director, but first we describe some additional generic options that apply only to directors (and not to routers). These can be divided into three of the same categories that were used in the previous chapter:

- Those that set conditions for the running of the director.
- Those that change what happens after a succesful run.
- Those that add data to an address that is accepted by the driver, for use when that address is delivered.

Conditional Running of Directors

In addition to the options described in the section "Conditional Running of Routers and Directors," in Chapter 6, *Options Common to Directors and Routers*, there are some other conditional options that apply only to directors (not to routers).

Local Part Prefixes and Suffixes

There are a number of common situations in which it is helpful to be able to recognize a prefix or a suffix on a local part, and to handle the affix and the remainder of the local part independently. The `prefix` and `suffix` options provide this facility, and are introduced in the section "Closed Mailing Lists," in Chapter 5, *Extending the Delivery Configuration*. They can be set to colon-separated lists of strings. If either is set, the director is skipped unless the local part starts with, or ends with (respectively), one of the given strings.

Another example of the use of a prefix is to provide a way of bypassing users' *.forward* files. For example, you can do this with the following director:

```
real_users:
  driver = localuser
  prefix = real-
  transport = local_delivery
```

If a local part begins with `real-`, this director is run, and if the rest of the local part is a login name, the address is accepted and directed to the **local_delivery** transport. Local parts that do not start with the prefix are not handled by this director, and are therefore passed on.

Processing of `prefix` and `suffix` happens after checking the `local_parts` condition. Therefore, the local part that is checked against `local_parts` is the full local part, including any prefix or suffix. If you want to select a director based on a partial local part, you can use a regular expression, or make use of the `conditions` option to do more complicated processing. For example, if you want to run a director for local parts that start with `owner-`, but only if the rest of the local part is listed in some file, you could use:

```
condition = \
  ${if match {$local_part}{^owner-(.*)\$}\
    {${lookup{$1}lsearch{/some/file}{yes}{no}}}\
    {no}}
```

The condition uses a regular expression to check that the local part starts with `owner-`, and to extract the remainder into $1. This is then used as the key for a file lookup.

The use of a prefix or suffix can be made optional by setting `prefix_optional` or `suffix_optional` as appropriate. In these cases, the director is run whether or not the affix matches, but if there is an affix, it is removed while running the director.

A limited form of wildcard is available for prefixes and suffixes. If a prefix begins with an asterisk, it matches the longest possible sequence of arbitrary characters at the start of the local part. Similarly, if a suffix ends with an asterisk, it matches the longest possible sequence at the end. There is an example of how this might be used in the section "Multiple User Addresses," in Chapter 5, using the option:

```
    suffix = -*
```

If you use this kind of wildcard, you need to choose a character that never appears in nonaffixed local parts as a separator, such as the hyphen in the previous example.

If both `prefix` and `suffix` are set for a director, both conditions must be met if they are not optional. Care must be taken if wildcards are used in both a prefix and a suffix on the same director. Different separator characters must be used to avoid ambiguity.

Control of EXPN

There is one other generic option that applies a condition to the running of a director. It is called `expn` and concerns the use of the SMTP `EXPN` command. This command is not used for mail delivery, but instead requests the expansion of an address, to show the result of aliasing or forwarding. `EXPN` has fallen out of favor recently, and many sites consider it to be a privacy exposure. For this reason, Exim permits it only from hosts that match an item in the `smtp_expn_hosts` main configuration option.

If the option is turned off for any director by including `no_expn` in the configuration file, the director is skipped when expanding an address as a result of processing an `EXPN` command. If you permit `EXPN` at all, you might, for example, want to turn the option off on a director for users' *.forward* files, while leaving it on for the system alias file. This option is specific to directors because `EXPN` applies only to local addresses.

Optimizing Single-Level Aliasing

When an **aliasfile**, **forwardfile**, or **smartuser** director generates a "child" address, it is normally processed from scratch, just like the original addresses in a message, and it may itself give rise to further aliasing or forwarding. However, sometimes an administrator knows that it is pointless to reprocess such addresses with the same director again. For example, if an alias file translates real names into login IDs there is no point searching the alias file a second time, especially if it is large.

The `new_director` option can be set to the name of any director instance. It causes any local addresses generated by the current director to be processed by starting at the named director instead of at the first director. The named director can be any configured director. For example, with an alias file containing the following:

```
J.Caesar:    jules
M.Anthony:   mark
```

the performance of the standard configuration could be improved by adding:

```
new_director = userforward
```

to the **system_aliases** director. The local part *J.Caesar* would be turned into *jules* by this director, but then processing of *jules* would start at the director called **userforward**, instead of at the first defined director. In the standard configuration, this director immediately follows **system_aliases**, so an unnecessary alias lookup is avoided. The `new_director` option has no effect if the director in which it is set does not generate new addresses, or if such addresses are not in local domains.

Adding Data for Use by Transports

Whenever a local transport is run, a current directory is set, and there may be a home directory value in $home. In most cases, you don't have to worry about these, but occasionally it is necessary to change the default settings. The options `current_directory` and `home_directory` can be set on any local transport or on any director. A setting on the transport overrides a setting on a director.

The director options associate values with any address that a director sends to a local transport, either as an original address or because it generates a delivery to a file or a pipe. During the delivery process (that is, at transport time), the option strings are expanded and set as the current directory or home directory, respectively, unless overridden by settings on the transport. Because the expansions do not take place until the transport is run, the value of `home_directory` is *not* available in $home while the director is running.

The **localuser** director and the **forwardfile** director with `check_local_user` use the user's home directory as the ultimate default for both these directories.

The aliasfile and forwardfile Directors

The **aliasfile** and **forwardfile** directors have many things in common. They each use the local part to find a list of new addresses or processing instructions. Later, in the sections on each director, we discuss how they obtain the data that they use. In this section, we consider first the types of item that may appear in the list. Then we describe the options that are common to both **aliasfile** and **forwardfile**. These are the ones that are concerned with handling the items in the list. In addition, each director has its own options controlling the way the list is obtained.

Items in Alias and Forward Lists

The items in an alias or forwarding list are separated by newlines or commas.* Empty items are ignored. If an item is entirely enclosed in double quotes, these are removed. Otherwise, double quotes are retained because some forms of mail address require their use (but never to enclose the entire address). In the following description, "item" refers to what remains after any surrounding double quotes have been removed.

Duplicate addresses

Exim removes duplicate addresses from the list of addresses to which it is delivering, so as to deliver just one copy to each unique address. This also applies to any items in aliasing or forwarding lists. For example, if a message is addressed to both *postmaster* and *hostmaster*, and these both happen to be aliased to the same person, a single copy of the message is delivered. This optimization does not apply to deliveries directed at pipes, provided the immediate parent addresses are different. Thus, if two different recipients of the same message happen to have set their *.forward* files to pipe to the same command,† two distinct deliveries are made. Within an alias file, a scheme such as:

```
localpart1: |/some/command
localpart2: |/some/command
```

does result in two different pipe deliveries, because the immediate parents of the pipes are distinct. However, an indirect aliasing scheme of the type:

```
pipe:       |/some/command
localpart1: pipe
localpart2: pipe
```

does a single delivery only because the intermediate local parts, which are the immediate parents of the pipe commands, are identical.

* Newlines cannot be present in alias lists obtained from linearly searched files, because they are removed by the continuation-line processing, but they can be present in lists obtained by other lookup methods.

† */usr/bin/vacation* is a common example.

Including the incoming address in a list

It is safe for a local part to generate itself, because Exim has a general mechanism for avoiding loops. A director or router is automatically skipped if any ancestor of the current address is identical to it and was processed by that director. Thus, a user with login name *spqr* who wants to preserve a copy of mail and also forward it somewhere else, can set up a *.forward* file such as:

```
spqr, spqr@st.elsewhere.example
```

without provoking a loop, because when *spqr* is processed for the second time, the **forwardfile** director is skipped. A backslash before a local part with no domain is permitted; for example:

```
\spqr, spqr@st.elsewhere.example
```

This is for compatibility with other MTAs, but is not necessary in order to prevent a loop.* The presence or absence of a backslash can, however, make a difference when there is more than one local domain. If `qualify_preserve_domain` is set for the director, a local part without a domain is qualified with the domain of the incoming address whether or not it is preceded by a backslash, but if `qual-ify_preserve_domain` is not set, a local part without a leading backslash is qualified with Exim's `qualify_recipient` value, independently of the incoming domain.

A bad interaction between aliases and forwarding

Care must be taken if there are alias names for local users who might have *.forward* files. For example, if the system alias file contains:

```
Sam.Reman: spqr
```

then:

```
Sam.Reman, spqr@reme.elsewhere.example
```

in *spqr*'s *.forward* file fails on an incoming message addressed to *Sam.Reman*; the incoming local part is turned into *spqr* by aliasing, and then the *.forward* file turns it back into *Sam.Reman*. When this "grandchild" address is processed, the **aliasfile** director is skipped in order to break the loop, because it has previously directed *Sam.Reman*. This causes *Sam.Reman* to be passed on to subsequent directors, which probably cannot handle it. The *.forward* file should really contain:

```
spqr, spqr@reme.elsewhere.example
```

but because this is such a common user error, the `check_ancestor` option exists to provide a way around it. How this works is described in the section "Options Common to aliasfile and forwardfile," later in this chapter.

* A backslash at the start of an item that is a qualified address (that is, with a domain) is not special; there are valid RFC 822 addresses that start with a backslash.

Nonaddress alias and forward items

The following types of nonaddress items may appear in a list generated by either aliasing or forwarding:

- An item is interpreted as a pathname if it begins with `/` and does not parse as a valid RFC 822 address that includes a domain. For example:

  ```
  /home/world/minbari
  ```

  is treated as a filename, but the following:

  ```
  /s=molari/o=babylon/@x400gateway.example
  ```

  is treated as an address. If a generated path is */dev/null*, delivery to it is bypassed at a high level, and the log entry shows `**bypassed**` instead of a transport name. This avoids the need to specify a user and group, which are necessary for a genuine delivery to a file. When the file name is not */dev/null*, either the director or the transport must specify a user and group under which to run the delivery.

- An item is treated as a pipe command if it begins with | and does not parse as a valid RFC 822 address that includes a domain. Either single or double quotes can be used for enclosing the individual arguments of the pipe command; no interpretation of escapes is done for single quotes. If the command contains a comma character, it is necessary to put the whole item in double quotes, as shown in:

  ```
  "|/some/command ready,steady,go"
  ```

  since items are terminated by commas. Do not, however, quote just the command. An item such as:

  ```
  |"/some/command ready,steady,go"
  ```

  is interpreted as a pipe with a rather strange command name, and no arguments.

- If an item takes the form:

  ```
  :include:path name
  ```

  a list of further items is taken from the given file and included at that point. The items in the file are separated by commas or newlines. If this is the first item in an alias list in a linearly searched file, a colon must be used to terminate the alias name, as otherwise the first colon is taken as the alias terminator, and the item is not recognized. This example is incorrect:

  ```
  eximlist   :include:/etc/eximlist
  ```

It must be written like this:

```
eximlist:  :include:/etc/eximlist
```

Items read from an included file are not subject to string expansion, even when the `expand` option is set on an **aliasfile** director. The use of `:include:` can be disabled by setting `forbid_include` on the director.

Nonaddress alias-only items

There are some additional special item types that can appear in lists generated by aliasing, but not by forwarding:

- Sometimes you want to throw away mail to a particular local part. An alias entry with no addresses causes Exim to generate an error, so that cannot be used. However, a special item that may appear in an alias file is:

  ```
  :blackhole:
  ```

  which does what its name implies. No delivery is done for it, and no error message is generated. This used to be more efficient than directing a message to */dev/null* because it happens at directing time, and also there was no need to specify a user and group to run the transport process for delivery to a file. However, in all but very old versions of Exim, */dev/null* is now recognized specially, and handled in essentially the same way.

- An attempt to deliver to a particular local part can be deferred or forced to fail by aliasing the local part to:

  ```
  :defer:
  ```

  or

  ```
  :fail:
  ```

  respectively. During message delivery, an alias containing `:fail:` causes an immediate failure of the incoming address, whereas `:defer:` causes the message to remain on the queue so that a subsequent delivery attempt can happen at a later time. If an address is deferred for too long, it will ultimately fail, because normal retry rules apply.

  When a list contains one of these items, any prior items in the list are ignored. Text that follows `:defer:` or `:fail:` is placed in the error message that is associated with the failure. A comma does not terminate the error text, but a newline does.* For example, an alias file might contain these lines:

* Newlines are not normally present in alias expansions. In `lsearch` lookups, they are removed as part of the continuation process, but they may exist in other kinds of lookup and in included files.

```
x.employee:  :fail:  Gone away, no forwarding address
j.caesar:    :defer: Mailbox is being moved today
```

In the case of an address that is being verified for the SMTP `RCPT` or `VRFY` commands, the text is included in the SMTP error response, which uses a 451 code for a deferral and 550 for a failure. If `:fail:` is encountered while a message is being delivered, the text is included in the bounce message that Exim generates; the text for `:defer:` appears in the log line for the deferral, but is not otherwise used.

- Sometimes it is useful to use a search type with a default for aliases, as in this example director we used for virtual domains:

```
virtuals:
  driver = aliasfile
  domains = cdb;/etc/virtuals
  file = /etc/$domain.aliases
  search_type = lsearch*
  no_more
```

However, there may be a need for exceptions to the default. These can be handled by aliasing them to:

```
:unknown:
```

For example:

```
*:          postmaster@virt3.example
postmaster: pat@dom5.example
jill:       jkr@dom4.example
jack:       :unknown:
```

This differs from `:fail:` in that it causes **aliasfile** to decline, so the address is offered to the next director, whereas `:fail:` forces directing to fail immediately without running any more directors.

All four of these special items can be disabled by setting `forbid_special` in the **aliasfile** configuration.

Options Common to aliasfile and forwardfile

A number of options that are common to both **aliasfile** and **forwardfile** are described in this section.

Checks on file attributes

You can request Exim to carry out checks on the ownership and mode of files used for aliasing or forwarding. In the standard configuration, each user's *.forward* file must be owned by that user, but there is no default set up for alias files.

Lists of permitted owners and groups can be given in the `owners` and `owngroups` options, and `modemask` specifies mode bits that must *not* be set. For example:

```
modemask = 007
owners = mail : root
owngroups = mail : root
```

specifies that the file must be owned by *mail* or *root*, and that none of the "other" access bits must be set. The default value for `modemask` is 022, which requires that the file not be writable by anyone other than the owner.

If the file's ownership or group ownership is incorrect, delivery is deferred, and the message is frozen. If the mode bits are incorrect, a **forwardfile** director just defers delivery, but an **aliasfile** director freezes the message.

When an **aliasfile** director uses a query-style database lookup (which does not name a specific file), these options are ignored.

Ancestor checking

In the section "Items in Alias and Forward Lists," earlier in this chapter, the rule Exim uses to prevent looping in the directors was discussed, and it was pointed out that a system alias file containing:

```
Sam.Reman: spqr
```

combined with a *.forward* file for *spqr* containing:

```
Sam.Reman, spqr@reme.elsewhere.example
```

did not work, because the alias *Sam.Reman* could be turned into the username *spqr* only once. This is such a common mistake that an option to get round it exists.

When `check_ancestor` is set, if a generated address is the same as any ancestor of the current address, it is not used, but instead the current address is passed on to subsequent directors. For the earlier problem example, if `check_ancestor` is set on the **forwardfile** director, it stops it turning *spqr* back into *Sam.Reman*. Instead, *spqr* is passed on to the next director.

The default configuration sets `check_ancestor` on the director that handles user's *.forward* files so that it is more likely to "do what the user meant."

Transports for pipes and files

When an aliasing or forwarding director generates deliveries directly to pipes or files, it is necessary to define the transports that are to be used by setting pipe_transport and file_transport, respectively. For example, the default Exim configuration handles system aliases by means of this director:

```
system_aliases:
  driver = aliasfile
  file = /etc/aliases
  search_type = lsearch
  file_transport = address_file
  pipe_transport = address_pipe
```

The presence of the final two options means that aliases such as:

```
fileit:    /some/file
pipeit:    |/some/command
```

are handled by the **address_file** and **address_pipe** transports, respectively.* The first of these is normally an **appendfile** transport that adds new messages onto the end of a mailbox file.

Disabling pipes and files

There are two options that lock out the use of pipes and files by an aliasing or forwarding director: forbid_pipe disallows pipe commands, and forbid_file disallows directing to a filename. If a disallowed item is encountered, delivery of the address fails. These options are often used on **forwardfile**, to restrict what users are permitted to put in their *.forward* files, and in fact there are some more options beginning with forbid_ that are specific to that director (see the section "Disabling Certain Features," later in this chapter).

Unqualified addresses

If an item that is generated by aliasing or forwarding consists of a local part only, and is not preceded by a backslash, the domain that is added is the default qualifying domain for the configuration (that is, the value of the global option qualify_domain). However, if qualify_preserve_domain is set, the domain of the incoming address is used instead. The option makes a difference only in cases where more than one local domain is in use.

Rewriting generated addresses

Generated addresses are normally rewritten according to the configured rewriting rules (see Chapter 14, *Rewriting Addresses*), but if you don't want this to happen, you can set no_rewrite in the director's configuration.

* Either the director or the transports also need to set a user under which the delivery is to run.

One-time aliasing and forwarding

When Exim has to retain a message for later delivery because it could not complete all the deliveries at the first attempt, it does not normally save the results of any aliasing or forwarding that was done. The next time it tries to deliver, each original recipient address is reprocessed afresh. This has the advantage that errors in alias lists and forward files can be corrected, but it has one disadvantage in the case of mailing lists that change frequently.

If one message is taking a very long time to be delivered to one subscriber, and new addresses are added to the list in the meantime, the new subscribers receive a copy of the old message, even though it dates from before their subscription. This can be avoided by setting the `one_time` option on the director that expands the mailing list. This changes Exim's behavior so that, after a temporary delivery failure, it adds the undelivered "child" addresses to the top-level list of recipients, and marks the original address as delivered. Thus, subsequent changes to the mailing list no longer affect this message.

The original top-level address is remembered with each of the generated addresses, and is output in log messages. However, intermediate parent addresses are not recorded. This makes a difference to the log only if `log_all_parents` is set. It is expected that `one_time` will typically be used for mailing lists, where there is normally just one level of expansion.

Setting `one_time` is possible only when there are no pipe or file deliveries in the alias or forwarding list, because it is not possible to turn these into top-level addresses. For this reason, Exim insists that `forbid_pipe` and `forbid_file` be set when `one_time` is set.

Missing include files

If an external file is included in a list, for example, by an alias such as:

```
eximlist:  :include:/etc/eximlist
```

Exim freezes delivery of the message if it cannot open the file, on the grounds that this is a serious configuration error. However, in some circumstances (for example, an NFS-mounted file), a file may sometimes be temporarily absent, and freezing is not appropriate. If `no_freeze_missing_include` is set, Exim just defers delivery, without freezing, if it cannot open an include file.

Syntax errors in alias or forward lists

If Exim discovers a syntax error in an alias or forward list, it defers delivery of the original address. This is the safest action to take. However, in some circumstances this may not be appropriate. For example, if a mailing list is being maintained by some automatic subscription process, you don't want one subscriber's typo to hold up deliveries to the rest of the list.

If `skip_syntax_errors` is set, a malformed item is skipped, and an entry is written to the main log. If `syntax_errors_to` is also set, a mail message is sent to the address it contains, giving details of the failing address(es). Often it will be appropriate to set `syntax_errors_to` to the same address as `errors_to` (the address for delivery failures). If `syntax_errors_text` is set, its contents are expanded and placed at the head of the error message.

Telling users about broken .forward files

Users often introduce syntax errors into their *.forward* files. They also often test them from their own accounts, usually several times when they observe the messages are not getting through. Using `skip_syntax_errors`, it is possible to deliver error messages into such users' mailboxes, thus reducing the postmaster load. First, you must arrange a way of delivering that bypasses user *.forward* files. A director that does this is explained in the section "Conditional Running of Directors," earlier in this chapter:

```
real_users:
  driver = localuser
  prefix = real-
  transport = local_delivery
```

The setting of `prefix` means that this director is skipped unless the local part is prefixed with *real-*. If it is defined before the **forwardfile** director, it picks off such local parts and sets up a local delivery, thereby bypassing any forwarding that might exist.

With this in place, `skip_syntax_errors_to` can be used on the **forwardfile** director to send a message to the user's inbox. Because we want to include newlines in the text string, it is given inside double quotes. When the value of an Exim option is quoted like this, a backslash inside the quotes is interpreted as an escape character: which provides a means of coding nonprinting characters. In particular, `\n` becomes a newline character.

```
userforward:
  driver = forwardfile
  file = .forward
  skip_syntax_errors
  syntax_errors_to = real-$local_part@$domain
  syntax_errors_text = "\
    This is an automatically generated message. \
    An error has been found\n\
    in your .forward file. Details of the error \
    are reported below. While\n\
    this error persists, messages addressed to \
    you will be delivered into\n\
        your normal mailbox and you will receive a \
        copy of this message for\n\
        each one."
```

If a syntax error is encountered, the failing address is skipped, and the warning message is sent to the user's mailbox, using the *real-* prefix to bypass forwarding. A final cosmetic touch to this scheme is to rewrite the address in the warning message's headers so as to remove the *real-* prefix, using a rewriting rule such as this:

```
    ^real-([^@]+)@  $1@$domain  h
```

Exim's address rewriting facilities are described in Chapter 14; the simple rule shown here rewrites addresses in header lines (leaving envelopes untouched) by removing `real-` from the start of the local part.

Summary of Options Common to aliasfile and forwardfile

The options that are applicable to both **aliasfile** and **forwardfile** are summarized in this section:

`check_ancestor` (Boolean, default = false)
: This option is concerned with handling generated addresses that are the same as some address in the list of aliasing or forwarding ancestors of the current address. When it is set, if a generated address is the same as any ancestor, it is not used, but instead a copy of the current address is passed on to subsequent directors. It is not commonly set on **aliasfile**.

`directory_transport` (string, default = unset)
: A director sets up a delivery to a directory when a pathname ending with a slash is specified as a new "address." The transport used is specified by this option, which, after expansion, must be the name of a configured transport.

`file_transport` (string, default = unset)
: A director sets up a delivery to a file when a pathname not ending in a slash is specified as a new "address." The transport used is specified by this option, which, after expansion, must be the name of a configured transport.

`forbid_file` (Boolean, default = false)
: If this option is true, the director may not generate an item that specifies delivery to a local file or directory. If it attempts to do so, a delivery failure occurs.

`forbid_include` (Boolean, default = false)
: If this option is true, the use of the special `:include:` item is not permitted, and if one is encountered, the message is frozen.

`forbid_pipe` (Boolean, default = false)
: If this option is true, the director may not generate an item that specifies delivery to a pipe. If it attempts to do so, a delivery failure occurs.

`freeze_missing_include` (Boolean, default = true)
: If a file named by the `:include:` mechanism fails to open, delivery is frozen if this option is true. Otherwise, delivery is just deferred. Unsetting this option can be useful if included files are NFS-mounted and may not always be available.

`modemask` (octal-integer, default = 022)
: This specifies mode bits that must not be set for an alias or forward file. If they are set, delivery is deferred.

`one_time` (Boolean, default = false)
: If `one_time` is set, and any addresses generated by the director fail to deliver at the first attempt, the failing addresses are added to the message as "top level" addresses, and the parent address that generated them is marked "delivered." Thus, forwarding or aliasing does not happen again at the next delivery attempt. To ensure that the director generates only addresses (as opposed to pipe or file deliveries), `forbid_file` and `forbid_pipe` must also be set.

`owners` (string list, default = unset)
: This specifies a list of permitted owners for an alias or forward file. In the case of a **forwardfile** director that is configured to check for a local user, that user is automatically added to the list. If `owners` is unset and there is no local user involved, no check on the ownership is done. Otherwise, if the file is not owned by a user in the list, delivery is deferred and the message is frozen.

`owngroups` (string list, default = unset)
: This specifies a list of permitted groups for an alias or forward file. In the case of a **forwardfile** director configured to check for a local user, that user's default group is automatically added to the list. If `owngroups` is unset and there is no local user involved, no check on the file's group is done. If the file's group is not in the list, delivery is deferred and the message is frozen.

`pipe_transport` (string, default = unset)
: A director sets up a delivery to a pipe when a string starting with a vertical bar character is specified as a new "address." The transport used is specified by this option, which, after expansion, must be the name of a configured transport.

`qualify_preserve_domain` (Boolean, default = false)
: If this is set and an unqualified address (one without a domain) is generated, it is qualified with the domain of the incoming address instead of the value of `qualify_recipient`.

`rewrite` (Boolean, default = true)
If this option is set false, addresses generated by the director are not subject to address rewriting. Otherwise, they are treated like new addresses, and the rewriting rules (see Chapter 14) are applied to them.

`skip_syntax_errors` (Boolean, default = false)
If `skip_syntax_errors` is set, a malformed address that causes a parsing error is skipped, and an entry is written to the main log. This may be useful for mailing lists that are automatically managed.

`syntax_errors_text` (string, default = unset)
See `syntax_errors_to`.

`syntax_errors_to` (string, default = unset)
This option applies only when `skip_syntax_errors` is set. If any addresses are skipped because of syntax errors, a mail message is sent to the address specified by `syntax_errors_to`, giving details of the failing addresses. If `syntax_errors_text` is set, its contents are expanded and placed at the head of the error message.

The aliasfile Director

The **aliasfile** director expands local parts by consulting a file or database of aliases. An incoming local part is looked up, and the result is a list of one or more replacement addresses, filenames, pipe commands or certain special items, as described in the section "Items in Alias and Forward Lists," earlier in this chapter. The lookup is performed using Exim's standard lookup mechanisms, as described in Chapter 16, *File and Database Lookups*. This means that several different file formats, or databases such as NIS, LDAP, or MySQL, can be used to store alias lists. Furthermore, the standard lookup defaulting mechanism can be used if a default is required.

Exim is not limited to a single alias file; you can have as many **aliasfile** directors as you like, with each searching a different set of data. However, there is nothing special about such a sequence of directors; as soon as any one of them accepts an address, processing that address ceases.

Unless Exim's `locally_caseless` option has been set false, local parts are forced to lowercase in addresses that are directed. Thus, the keys in alias files should normally be in lowercase. For linearly searched files, this isn't in fact necessary, because the searching is done in a case-independent manner, but it is relevant for other forms of alias lookup.

Specifying the Lookup

Most of the options specific to **aliasfile** control the kind of lookup that it does. The lookup type must be specified in search_type. If it is a single-key type (lsearch, dbm, cdb, or nis), file must be set to the name of the file to be searched. For example:

```
search_type = cdb
file = /etc/aliases.cdb
```

If **aliasfile** cannot open the file because it does not exist, delivery is normally deferred, but if the optional option is set, the address is passed on to the next director instead. By default, the key that is looked up is just the local part (for example, *postmaster*), but if include_domain is set, the full address is used. This makes it possible to hold aliases for several domains in a single file such as this:

```
postmaster@domain1:   jill@domain1
postmaster@domain2:   jack@domain2
```

It is not possible to mix the two types in the same file and access it with a single director, but there is no reason why two different directors, one with include_domain and one without, should not search the same file. For example:

```
alias1:
  driver = aliasfile
  file = /mixed/file
  search_type = cdb
  include_domain

alias2:
  driver = aliasfile
  file = /mixed/file
  search_type = cdb
```

The first director searches for the full address; if it is not found, the second director searches for just the local part. Exim's caching mechanisms keep the file open from one director to the next, so this might even be slightly more efficient than using two different files.

If a query-style (database) lookup type is used, either a single query must be given in the query option, or a colon-separated list of queries must be given in the queries option. For example:

```
system_aliases:
  driver = aliasfile
  query = [name=${quote_nisplus:$local_part}],aliases.org_dir:address
  search_type = nisplus
```

contains a single query. If you wanted all unknown aliases to default to postmaster, you could replace this with:

```
system_aliases:
  driver = aliasfile
  queries = \
    [name=${quote_nisplus:$local_part}],aliases.org_dir::address : \
    [name=postmaster],aliases.org_dir::address
  search_type = nisplus
```

First, it would look up the local part, and if that was not found, it would try looking up *postmaster*. Notice that the colon in each of the queries has to be doubled to avoid its being taken as a list separator. If a query cannot be completed for some reason (for example, a database is offline), the director causes delivery to be deferred.

Expanding a List of Aliases

The list of items obtained from the file or database lookup is interpreted as described in the section "Items in Alias and Forward Lists," earlier in this chapter. However, if `expand` is set, the data is passed through the string expansion mechanism before it is interpreted. For example, consider this entry in an alias file:

```
somelist:   :include:/etc/lists/$domain
```

Suppose Exim is processing the local address *somelist@simple.example*. The **aliasfile** director finds the alias, and reads this aliasing data:

```
:include:/etc/lists/$domain
```

In the default state (`expand` not set), this is interpreted as a request to include the contents of the file whose name is */etc/lists/$domain*. However, if `expand` is set, the string is expanded before it is interpreted, so the file that is included is */etc/lists/simple.example*.

Specifying a Transport for aliasfile

This section describes a different mode of operation that applies to **aliasfile** when it is configured with a setting of the `transport` option.

The `transport` option must not be specified for **aliasfile** when it is fulfilling the traditional aliasing function of replacing the original address with one or more new addresses that are each going to be processed independently.

When a transport *is* specified, the director behaves quite differently, and doesn't really "alias" at all. Its lookup facilities are used as a means of validating the incoming address, but if it is successful, the message is directed to the given transport, *while retaining the original address*. The data that is returned from the lookup is not used. For example, a file containing a list of cancelled users can be used to direct messages addressed to them to a particular transport by a director like this:

```
cancelled_users:
  driver = aliasfile
  transport = cancelled
  file = /etc/cancelled_users
  search_type = lsearch
```

The file could contain lines such as this:

```
x.employee: gone away, no forwarding address
j.retired:  gone fishing
```

The **cancelled** transport could run a script, or it could be an **autoreply** transport that sends a message back to the sender.

Another common use of **aliasfile** with a transport setting is for handling local deliveries without reference to */etc/passwd* or other password data. This makes it possible to deliver to user mailboxes on a host where the users do not have login accounts. Local parts are validated by using **aliasfile** to look them up in a file or database, which can also be used to hold information for use during delivery (for example, the uid to use, or the location of the mailbox).

The use of a transport setting with **aliasfile** originated in the early days of Exim when the **smartuser** director was not as flexible as it is now. These days, **smartuser** can be used to provide the same features, and because the use of a transport with **aliasfile** is confusing, the facility may be abolished in some future release.

Take care not to confuse the generic `transport` option, which has the special behavior just mentioned, with `file_transport` and `pipe_transport` (as described in the section "Transports for pipes and files," earlier in this chapter), which provide entirely different facilities.

Summary of aliasfile Options

The options that are specific to **aliasfile** are summarized in this section; of course, **aliasfile** also accepts those options that are common to both **aliasfile** and **forwardfile**, as described earlier in the section "Options Common to aliasfile and forwardfile."

`expand` (Boolean, default = false)

If this option is set true, the text obtained by looking up the local part is passed through the string expansion mechanism before being interpreted as a list of alias items.

Addresses that are subsequently added by means of the "include" mechanism are *not* expanded.

`file` (string, default = unset)
This option specifies the name of the alias file, and it must be set if `search_type` specifies a single-key lookup; if it does not, an error occurs. The string is expanded before use; if expansion fails, Exim panics. The resulting string must be an absolute path for lookups that read regular files.

`forbid_special` (Boolean, default = false)
If this option is true, the special items `:defer:`, `:fail:`, `:blackhole:`, and `:unknown:` are not permitted to appear in the alias data, and if one is encountered, delivery is deferred.

`include_domain` (Boolean, default = false)
Setting this option true causes the key that is looked up to be *local-part@domain* instead of just *local-part*. By this means, a single file can be used to hold aliases for many local domains. This option has no effect if the search type specifies a query-style lookup.

`optional` (Boolean, default = false)
For a single-key lookup, if the file cannot be opened because it does not exist (the `ENOENT` error) and this option is set, the director declines to handle the address. Otherwise any failure to open the file causes an entry to be written to the log and delivery to be deferred.

For a query-style lookup, setting `optional` changes the behavior when a lookup cannot be completed (for example, when a database is offline). Without `optional`, the delivery is deferred; with `optional`, the director declines, and so the address is offered to the next director.

`queries` (string, default = unset)
This option is an alternative to `query`; the two options are mutually exclusive. The difference is that `queries` contains a colon-separated list of queries, which are tried in order until one succeeds or defers, or all fail. Any colon characters actually required in an individual query must be doubled so that they aren't treated as query separators.

`query` (string, default = unset)
This option specifies a database query, and either this option or `queries` must be set if `search_type` specifies a query-style lookup; if neither is set, an error occurs. The query is expanded before use, and would normally contain a reference to the local part. For example:

```
search_type = nisplus
query = [alias=${lookup_nisplus:$local_part}],\
        mail_aliases.org_dir:expansion
```

could be used for a NIS+ lookup. Sometimes a lookup cannot be completed (for example, a NIS+ database might be inaccessible), and in this case, the director causes delivery to be deferred.

`search_type` (string, default = unset)
This option must be set to the name of a supported search type (`lsearch`, `dbm`, and so on), specifying the type of data lookup. Single-key search types can be preceded by `partial-` and/or followed by an asterisk. The former is not likely to be useful very often, but the latter provides a default facility. Note, however, that if two addresses in the same message provoke the use of the default, only one copy is delivered, but any added *Envelope-to:* header contains all the original addresses. Exceptions to the default can be set up by aliasing them to `:unknown:`.

The forwardfile Director

The **forwardfile** director expands the local part of an address by reading a list of new addresses, filenames, pipe commands, or certain other special items from a given file. There are two common cases: processing a user's *.forward* file (from which the director gets its name), and expanding a mailing list.

From Release 3.20, the data that **forwardfile** uses may alternatively be supplied as an expanded string in the configuration. This makes it possible to hold filtering instructions in databases such as LDAP or MySQL.

Contents of forwardfile Lists

The data that **forwardfile** processes can be a simple list of items, separated by commas or newlines. However, it is also possible to request that Exim process the data as an *Exim filter*, which means that it is interpreted in a more complex way. In particular, in a filter, conditions can be imposed on which deliveries are performed. If the `filter` option is set on the director, and the first line of the data starts with:

```
# Exim filter
```

it is interpreted as a filter rather than a simple list of addresses. The use of filters is described in Chapter 10, *Message Filtering*. Whether a file is interpreted as a filter or a plain list does not affect the other actions of **forwardfile**; these are just two different ways of setting up a list of delivery items.

If a forward file exists but is empty, or contains only blank lines and comment lines starting with `#`, Exim behaves as if it did not exist, and the director declines to handle the address. Note that this is not the case when the file contains syntactically valid items that happen to yield empty addresses (for example, items containing only RFC 822 address comments).

Specifying a .forward File

If the `file` option is set, it specifies that the data to be processed is the entire contents of the file. If the filename is not an absolute path, it is taken relative to the directory that is defined by the `file_directory` option. This on its own is not very useful, because you might as well use:

```
file = /usr/forwards/$local_part.forward
```

instead of:

```
file_directory = /usr/forwards
file = $local_part.forward
```

However, if the directory and filename are given separately, the existence of the directory is tested before trying to open the file, and if the directory appears not to exist, delivery is deferred. This distinguishes between the cases of a nonexistent file (where the director should decline to handle the address) and an unmounted NFS directory (where delivery should be deferred).

Specifying an Inline Forwarding List

If the `file` option is not set, the data to be processed must be specified by the `data` option.[*] This string is expanded; the result is treated as a forwarding list. The expansion makes it possible to obtain the list from an indexed file or database, by using a lookup expansion item. For simple lists of addresses, much the same effect can be obtained by using a **smartuser** director, but **forwardfile** must be used if the list consists of filtering instructions, because **forwardfile** is the only director that can handle them.

Checking Local Users

Because its most common use is in handling users' *.forward* files, **forwardfile** checks to see whether the local part is the login name of a local user, and if it is not, passes it on to the next director. This behavior can be disabled by setting `no_check_local_user`; this is normally required when using **forwardfile** to process mailing lists. When a local user has been found, the home directory is used as the default value for the `file_directory` option, and the username and group are implicitly included in `owners` and `owngroups`. Thus, the very simple configuration:

```
userforward:
  driver = forwardfile
  file = .forward
```

checks for a local user, finds the user's home directory, checks that the directory exists, and then looks for *.forward* inside it. Checks on the owner, group owner,

[*] Available only from Release 3.20 onward.

and mode of the file are carried out as described in the section "Options Common to aliasfile and forwardfile," earlier in this chapter, using the local user's uid and gid. However, for **forwardfile**, the group ownership is checked only if `check_group` is set.

A further check can be imposed by setting the `match_directory` option, which applies only when **forwardfile** has checked the local part for a local login name. The value of `match_directory` is expanded and matched against the name of the user's home directory. If there is no match, the address is passed on to the next director. This provides a way of skipping *.forward* file processing for logins that do not have accessible home directories. This is important, because otherwise failure to find the home directory causes delivery to be deferred.

For example, if the names of the home directories of all the regular users on a system begin with */home/* or */group/*, but there are other logins with mailboxes but without accessible home directories, the following:

```
match_directory = ^/(home|group)/
```

could be used to skip forward file processing for the special logins. This example uses a simple regular expression to check that the home directory begins with `/home/` or `/group/`. The matching process is the same as used for domain list items, and as well as a regular expression (as in this example), a string beginning with an asterisk, such as:

```
match_directory = */special
```

or even a lookup such as:

```
match_directory = lsearch;/list/of/homes
```

can be used.

Special Error Handling

If a forward file exists, but cannot be opened for reading, delivery is deferred, but this can be changed for two specific opening errors. If `ignore_eacces`* is set, a "permission denied" error is treated as if the file did not exist, so the address is passed to the next director. If `ignore_enotdir` is set, a "not a directory" error (something on the path is not a directory) is treated likewise. The first of these cases can arise when personal forward files are being read from NFS file systems that are mounted without root access.

* The spelling of this option derives from the `EACCES` error code.

Disabling Certain Features

There are some extra `forbid_` options, in addition to those that are common to both **forwardfile** and **aliasfile**, that disable the use of certain features in filter files. Filters have been mentioned briefly, but the full details are deferred until Chapter 10, so these options probably won't make much sense at a first reading. Just remember that there are ways of locking out certain filtering features, and come back to this section when you need it:

`forbid_filter_existstest`
: Disables the use of the `exists` condition in string expansions in filters.

`forbid_filter_lookup`
: Disables the use of lookup items in string expansions in filters.

`forbid_filter_logwrite`
: Disables the use of the *logwrite* command in filters.

`forbid_filter_perl`
: Disables the use of embedded Perl in string expansions in filters.

`forbid_filter_reply`
: Disables the use of the *reply* command in filters.

The first three options lock out direct access to other files from a filter file. This could be appropriate when the filter is run on a system to which its owner has no login access.

Embedded Perl is available only if Exim has been built to support it, and it is likely to be of use only to the system administrator; for added security it is probably a good idea to disable it in users' filter files.

The *reply* command allows the creation of automatic replies to incoming messages from within filter files. Even when it is enabled, it cannot be used unless `reply_transport` has been set to define the **autoreply** transport that is used to create the replies. For example, in the default Exim configuration, such a transport is defined as:

```
address_reply:
  driver = autoreply
```

and the **forwardfile** director for users' *.forward* files contains:

```
reply_transport = address_reply
```

to allow the *reply* command to make use of it.

Enabling Certain System Actions

Another option that relates to filter files is `allow_system_actions`. This enables the filter commands *fail* and *freeze*, which are normally permitted only in system filter files. It is not normally sensible to give end users access to these commands in their personal filter files. However, some installations run centrally managed filter files on behalf of individual users, and in these cases, the ability to freeze a message or fail an address can be useful.

The $home Variable

The $home expansion variable can be used in a number of local options for **forwardfile**. Its value depends on the value of the `check_local_user` and `file_directory` options, but it is independent of the order in which the options appear in the configuration file:

- If `check_local_user`is set and `file_directory` is unset, $home is set to the user's home directory when expanding the `file` option.
- If `check_local_user` is unset and `file_directory` is set, $home is set to the expanded value of `file_directory` when expanding the `file` option. If $home appears in `file_directory` itself, its substitution value is the empty string.
- If both `check_local_user` and `file_directory` are set, $home contains the user's home directory when expanding `file_directory`, but subsequently $home contains the value of `file_directory` when expanding the `file` option. Consider these settings:

  ```
  check_local_user
  file_directory = $home/mail
  file = $home/.forward
  ```

  If *spqr*'s home directory is */home/spqr*, `file_directory` would be set to */home/spqr/mail*, whereas `file` would be set to */home/spqr/mail/.forward* when processing *spqr*'s mail.
- If neither `check_local_user` not `file_directory` are set, $home is empty.

If the generic `require_files` option, or any other expanded option, contains $home, it takes the same value as it does when expanding the `file` option. This value is also used for $home if encountered in a filter file, and as the default value to pass with the address when a pipe or file delivery is generated.

Current and Home Directories

The values of the `current_directory` and `home_directory` generic options are not used during the running of **forwardfile**; they specify directories for use at transport time in the event that **forwardfile** directs an address to a file, pipe command, or autoreply.

If `home_directory` is not set, the directory specified by `file_directory` is used instead. If `file_directory` is also unset, the home directory obtained from `check_local_user` is used.

This rule can lead to a problem in installations where users' *.forward* files are not kept in their home directories. In such installations, both `check_local_user` and `file_directory` may be set. For example:

```
check_local_user
file_directory = /etc/forwardfiles
file = $local_part.forward
```

With this configuration, the default value for `home_directory` is */etc/forwardfiles*, as just described. However, what may be desired is the user's actual home directory. It is no good specifying:

```
home_directory = $home
```

because when `home_directory` is expanded, the value of $homeis the same as when `file` is expanded, and in this case, that is the contents of `file_directory`, as described in the previous section. A special string value is therefore provided for use in this case. If `home_directory` is set thus:

```
home_directory = check_local_user
```

it is converted into the user's home directory path. The same magic string can be used for `current_directory`.

Summary of forwardfile Options

The options that are specific to **forwardfile** are summarized in this section; of course, **forwardfile** also accepts those options that are common to both **aliasfile** and **forwardfile**, as described in the section "Options Common to aliasfile and forwardfile," earlier in this chapter.

`allow_system_actions` (Boolean, default = false)

Setting this option permits the use of `freeze` and `fail` in filter files. This should *not* be set on the director for users' *.forward* files, but can be useful if you want to run a systemwide filter for each address (as opposed to the system filter, which runs just once per message).

`check_group` (Boolean, default = false)
: The group owner of the file is checked only when this option is set. If `check_local_user` is set, the user's default group is permitted; otherwise the group must be one of those listed in the `owngroups` option.

`check_local_user` (Boolean, default = true)
: If this option is true, the local part of the address that is passed to this director is checked to ensure that it is the login of a local user. The director declines to handle the address if no local user is found. In addition, when this option is true, the string specified for the `file` option is taken as relative to the user's home directory if it is not an absolute path, and the `file_directory` option is not set.

  When `check_local_user` is set, the local user is always one of the permitted owners of the *.forward* file. In addition, the uid and gid obtained from the password data are used as defaults for the generic `user` and `group` options.

`data` (string, default = unset)
: This option must be set if `file` is not set. Its value is expanded and used as a list of forwarding items or filtering instructions.

`file` (string, default = unset)
: This option must be set if `data` is not set. The string is expanded before use. If expansion fails, Exim defers the address and freezes the message. The expanded string is interpreted as a single filename, and must start with a slash character unless `check_local_user` is true or a `file_directory` option is set. A nonabsolute path is interpreted relative to the `file_directory` setting if it exists; otherwise it is interpreted relative to the user's home directory.

`file_directory` (string, default = unset)
: The string is expanded before use. The option sets a directory path that is used if the `file` option does not specify an absolute path. Also, if **forwardfile** sets up a delivery to a file or a pipe command and the `home_directory` option is not set, the directory specified by `file_directory` is passed to the transport as the home directory. If `file_directory` is also unset, the home directory obtained from `check_local_user` is the home address during delivery.

`filter` (Boolean, default = false)
: If this option is set, and the *.forward* file starts with the text:

```
# Exim filter
```

  it is interpreted as a set of filtering commands instead of a list of forwarding addresses. Details of the syntax and semantics of filter files are described in Chapter 10.

`forbid_filter_existstest` (Boolean, default = false)
: If this option is true, string expansions in filter files are not allowed to make use of the `exists` condition.

`forbid_filter_logwrite` (Boolean, default = false)
: If this option is true, use of the logging facility in filter files is not permitted. This is in any case available only if the filter is being run under some unprivileged uid, which is normally the case for ordinary users' *.forward* files.

`forbid_filter_lookup` (Boolean, default = false)
: If this option is true, string expansions in filter files are not allowed to make use of `lookup` items.

`forbid_filter_perl` (Boolean, default = false)
: This option is available only if Exim is built with embedded Perl support. If it is true, string expansions in filter files are not allowed to make use of the embedded Perl support.

`forbid_filter_reply` (Boolean, default = false)
: If this option is true, this director may not generate an automatic reply message. If it attempts to do so, a delivery failure occurs. Automatic replies can be generated only from filter files, not from traditional *.forward* files.

`ignore_eacces` (Boolean, default = false)
: If this option is set and an attempt to open the *.forward* file yields the `EACCES` error (permission denied), **forwardfile** behaves as if the file did not exist, and passes the address on to the next director.

`ignore_enotdir` (Boolean, default = false)
: If this option is set and an attempt to open the *.forward* file yields the `ENOTDIR` error (something on the path is not a directory), **forwardfile** behaves as if the file did not exist, and passes the address on to the next director.

`match_directory` (string, default = unset)
: If this option is set with `check_local_user`, the user's home directory must match the given string. If it does not, the director declines to handle the address. The string is expanded before use. If the expansion fails, Exim panics, unless the failure was forced, in which case the director just declines.

: If the expanded string starts with an asterisk, the remainder must match the end of the home directory name; if it starts with a circumflex, a regular expression match is performed. In fact, the matching process is the same as used for domain list items and may include file lookups.

`reply_transport` (string, default = unset)

A **forwardfile** director sets up a delivery to an **autoreply** transport when a *mail* or *vacation* command is used in a filter file. The transport used is specified by this option, which, after expansion, must be the name of a configured transport.

The localuser Director

The **localuser** director checks whether the local part of the address that is being directed is the login of a local user. If it is, and if other conditions set by generic options such as `domains` are met, **localuser** accepts the address and sets up a transport for it.

Transports for localuser

The `transport` option must always be specified for **localuser**, unless the `verify_only` option is set, in which case the director is used only for checking addresses. The transport does not have to be a local transport; any transport can be used. For example, suppose that all local users have accounts on a central mail system, but have their mail delivered by SMTP to their individual workstations. On the central server, a **localuser** director such as this:

```
checklocals:
  driver = localuser
  transport = workstations
```

could be used with a remote transport such as this:

```
workstations:
  driver = smtp
  hosts = ${lookup{$local_part}cdb{/etc/workstations}{$value}fail}
```

where the file */etc/workstations* contains a mapping from username to workstation name.

When the transport is in fact a local one (the most common case), the user's uid and gid are set up by default to be used for the delivery process. If the `home_directory` option is unset, the user's home directory is passed to a local transport for use during delivery.

Checking the Home Directory

There is only one option that is specific to the **localuser** director, called `match_directory`. If it is set, the user's home directory must match the pattern in the option. If it does not, the director declines to handle the address, and it is offered to the following director. This provides a way of partitioning the local users by home directory. We saw an example of how this can be used in the section "Changing a Driver's Successful Outcome," in Chapter 6.

The string is expanded before use. If the expansion fails, Exim defers the address and freezes the message, unless the failure was forced, in which case the director just declines to handle the address. If the expanded string starts with an asterisk, the remainder must match the end of the home directory name; if it starts with a circumflex, a regular expression match is performed. In fact, the matching process is the same as used for domain list items, and may include file lookups.

The smartuser Director

The **smartuser** director makes no checks of its own on the address that is passed to it; it handles anything. One common use is to place an unconditional instance of **smartuser** as the last director to pick up all addresses that the other directors are unable to handle. However, `smartuser` is, of course, subject to the generic director options, so specific instances can be used for all addresses in certain domains, all local parts with certain prefixes or suffixes, specific local parts, or any other generic condition. There is an example of this in the section "Conditional Running of Routers and Directors," in Chapter 6.

The **smartuser** director operates in two different ways. It can either generate one or more new addresses, in a similar manner to **aliasfile** and **forwardfile**, or it can direct the incoming address to a specific transport.

Using smartuser to Generate New Addresses

If the generic `transport` option is not specified, **smartuser**'s `new_address` option must be set. This supplies a list of replacement addresses that are then handled by other drivers. The value of `new_address` is treated as if it were a line from an alias file, and so must consist of a comma-separated list of items. The special values `:blackhole:`, `:defer:`, and `:fail:` (but not `:include:`) may be used, and items may refer to files or pipes.

Unqualified addresses are qualified using the value of `qualify_recipient`, unless `qualify_preserve_domain` is set, in which case they take the domain of the incoming address. If any new address is a duplicate of any other address in the message, it is discarded.

This form of **smartuser** can be used for a number of special-purpose actions. Suppose you want to defer delivery to a specific local part for some reason (such as moving the mailbox). This director, placed at the top of the directors' configuration, does the job for the local part *notyet*:

```
defer:
  driver = smartuser
  local_parts = notyet
  new_address = :defer:
```

Another example came up on the Exim mailing list in which somebody wanted to delay deliveries for a specific local part by an hour. Adding the following:

```
condition = ${if < {$message_age}{3600}{yes}{no}}
```

to the earlier example achieves this effect. Deferring delivery in this way is subject to the normal retrying rules, so if it goes on long enough, the address is bounced.

Using smartuser to Direct to a Transport

If the generic `transport` option is specified, **smartuser** directs the message to that transport. For example:

```
unknown:
  driver = smartuser
  transport = unknown
```

directs the address to the **unknown** transport unconditionally. It is possible to change the address at the same time by setting `new_address`. For example:

```
unknown:
  driver = smartuser
  new_address = $local_part@plc.com.example
  transport = unknown
```

rewrites the envelope address by forcing a specific domain while retaining the old local part, and then directs the new address to the **unknown** transport. The original address is available to the transport via the expansion variables $original_local_part and $original_domain.

If the expansion of `new_address` is forced to fail, the director declines to handle the address, and consequently it is offered to the following director. Otherwise, unless `no_panic_expansion_fail` is set, an expansion failure is treated as a serious configuration error, and causes Exim to write a message to its panic log and exit immediately. New addresses are rewritten by Exim's normal rewriting rules (see Chapter 14) unless the `no_rewrite` option is set.

Exim normally checks for duplicates in the recipients of a message, and delivers only a single copy. However, when **smartuser** is used in this way, with both `transport` and `new_address` set, the new address is not checked for duplication.

If you want to have multiple deliveries for messages whose original recipients are aliased to the same final address, the only way to do it is by configuring **smartuser** appropriately. For example:

```
hostpost:
  driver = smartuser
  local_parts = hostmaster : postmaster
  transport = local_delivery
  new_address = dogsbody@example.com
  user = dogsbody
```

delivers mail for both *hostmaster* and *postmaster* into *dogsbody*'s mailbox, and if a message is addressed to both of them, two copies are written.

Summary of smartuser Options

The options that are specific to **smartuser** are summarized in this section:

`new_address` (string, default = unset)

When `transport` is set, this option specifies a single new address to replace the current one in the message's envelope when it is transported. The address must be qualified (that is, contain an `@` character).

When `transport` is not set, this option is treated like a line from an alias file. Any unqualified addresses it contains are qualified using the value of `qualify_recipient`, unless `qualify_preserve_domain` is set.

In both cases, new addresses are rewritten by Exim's normal rewriting rules unless the `rewrite` option is turned off.

`panic_expansion_fail` (Boolean, default = true)

If expansion of the `new_address` option fails (other than a forced failure), Exim panics if this option is set. Otherwise, the director declines, and the original address is offered to the next director.

`qualify_preserve_domain` (Boolean, default = false)

If this is set and an unqualified address (one without a domain) is found in a `new_address` list when **smartuser** is configured without a transport, the address is qualified with the domain of the incoming address instead of the value of `qualify_recipient`.

`rewrite` (Boolean, default = true)

If this option is set false, addresses specified by `new_address` are not subject to rewriting.

8

The Routers

Routers handle addresses whose domains are not local, and typically (though not necessarily) set up deliveries to remote hosts. The different routers use different methods to obtain a list of relevant hosts for the domain of the address they are handling. The hosts' IP addresses must also be looked up. Some examples of the **lookuphost** and **domainlist** routers appear in previous chapters, but there are also other routers. In this chapter, we discuss each of the following in detail:

domainlist
: A router that routes remote domains using locally supplied information.

ipliteral
: A router that handles "IP literal" addresses such as *user@[192.168.5.6]*. These are relics of the early Internet that are no longer in common use.

lookuphost
: A router that looks up remote domains in the DNS.

queryprogram
: A router that runs an external program in order to route a domain.

First, however, we cover some additional generic options that apply only to routers (and not to directors).

Timeouts While Routing

If a router times out while trying to look up an MX record or an IP address for a host, it normally causes delivery of the address to be deferred. However, if `pass_on_timeout` is set, the address is instead passed on to the next router, overriding `no_more`. This may be helpful for systems that are intermittently connected to the Internet, or those that want to pass to a smart host any messages that cannot be delivered immediately, as in this example:

```
lookuphost:
  driver = lookuphost
  transport = remote_smtp
  pass_on_timeout
  no_more

timedout:
  driver = domainlist
  transport = remote_smtp
  route_list = * smart.host.example byname
```

The first router looks up the domain in the DNS; if it is not found, the router declines, but because of `no_more`, no further routers are tried and the address fails. However, if the DNS lookup times out, the address is passed to the next router, which sends it to a smart host. We explain the details of how this **domainlist** router works later in this chapter.

A timeout is just one example of a temporary error that can occur while doing DNS lookups. All such errors are treated in the same way as a timeout, and this option applies to all of them.

Domains That Route to the Local Host

When a router is configured to set up a remote delivery, it generates a list of one or more hosts to which the message can be sent. The list contains the hosts in an order of preference (commonly obtained from MX records). If the local host appears other than at the start of the list, it is dropped from the list, along with any less preferred hosts, in order to avoid looping. If, on the other hand, the local host is the first host on the list, special action needs to be taken.*

This situation can arise as a result of a mistake in Exim's configuration (for example, the domain should be listed as local so that it is processed by the directors and not by the routers), or it may be an error in the DNS (for example, the lowest numbered MX record should not point to this host).

In a simple configuration, sending the message would cause a tight mail loop. Exim's default action is therefore to defer delivery and freeze the message to bring it to the administrator's attention. However, in more complicated situations, a different action may be required. What Exim does when a domain routes to the local host is controlled by the value of the `self` option. This can be set to one of a

* The test for the local host involves checking the IP address(es) of the supposedly remote host against the interfaces on the local host. If `local_interfaces`is set, only the interfaces it lists are tested.

number of descriptive words, the default being `freeze`. The option is relevant only when the router is configured to set up hosts for delivery; some routers can be configured in a mode that just rewrites the domain for further processing, and in these cases the `self` option is not relevant and is ignored.

Treating Domains Routed to self as Local

If `self` is set to `local`, the address is passed to the directors, as if its domain were a local domain. This can be used to treat any domain whose lowest MX record points to the host as a local domain, without having to list them all explicitly in `local_domains`. You would use a router such as this:

```
lookuphost:
  driver = lookuphost
  transport = remote_smtp
  self = local
```

This is a standard **lookuphost** router that looks up the domain in the DNS and routes to the **remote_smtp** transport in normal circumstances. However, if it ends up routing to the local host, the address is marked as local and passed to the directors. During subsequent directing and delivery the variable $self_hostname is set to the name of the host that was first in the list of hosts generated by the router (that is, the name that resolved to the local host).

Passing Domains Routed to self to the Next Router

If `self` is set to `pass`, the router declines, passing the address to the following router, and setting $self_hostname to the name of the host that was first in the list of hosts generated by the router; that is, the name that resolved to the local host. This setting of `self` overrides a setting of `no_more` on the router, so a combination of the following:

```
self = pass
no_more
```

ensures that only those addresses that routed to the local host are passed on. Without `no_more`, an address that was declined because the domain did not exist would also be offered to the next router. A corporate mail gateway machine might have this as its first router:

```
lookuphost:
  driver = lookuphost
  transport = remote_smtp
  self = pass
  no_more
```

Domains that resolve to remote hosts are routed to the **remote_smtp** transport in the normal way. Domains that are unrecognized are bounced, because `no_more` prevents them being offered to any subsequent routers, but domains that resolve to the local host are passed on because of the setting of `self`. Subsequent routers can then assume they are dealing with the set of domains whose DNS entries point to the local host.

Rerouting Domains Routed to self

If `self` is set to `reroute:` followed by a domain name, the domain is changed to the given domain, and the address is passed back to be reprocessed by the directors or routers, as appropriate. For example:

```
self = reroute: newdom.example.com
```

changes the domain to *newdom.example.com* and reprocesses the address from scratch. No rewriting of header lines takes place, but there is an alternative form that does cause header rewriting:

```
self = reroute: rewrite: newdom.example.com
```

In this case, any addresses in the header lines that contain the old domain are rewritten with the new one.

Failing Domains Routed to Self

If the `self` option is set to `fail`, the router declines, but the address is not passed to any following routers. Consequently, delivery fails and an error report is generated.*

Transporting Domains Routed to self

If the `self` option is set to `send`, the routing anomaly is ignored and the address is passed to the transport in the usual way. This setting should be used with extreme caution because of the danger of looping. For remote deliveries, it makes sense only in cases where the program that is listening on the TCP/IP port of the local host is not this version of Exim. That is, it must be some other MTA, or Exim with a different configuration file that handles the domain in another way.

Deferring Domains Routed to self

Finally, if the `self` option is set to `defer`, delivery of the address is deferred but the message is not frozen, so delivery will be retried at intervals. If this goes on long enough, the address will time out and be bounced.

* In earlier versions of Exim, `fail_soft` and `fail_hard` were used instead of `pass` and `fail`. The older settings are still recognized.

The lookuphost Router

The **lookuphost** router uses a standard system interface to look up the hosts that handle mail for a given domain. Normally, it uses the DNS resolver to find the information from the DNS, but it can be configured to use the operating system's host lookup function (which might consult */etc/hosts* or NIS as well as the DNS) instead. A transport must always be set for this router, unless `verify_only` is set. We have previously shown the most basic configuration of **lookuphost** as:

```
lookuphost:
  driver = lookuphost
  transport = remote_smtp
```

This uses the DNS according to the standard mail routing rules: it first looks for MX records for the domain; if found, they provide a list of hosts. If there are no MX records, it looks up address records for a host that has the domain's name.

If the router cannot find out whether the domain does or does not have any MX records (because of a timeout or other DNS failure), it cannot proceed, and delivery is deferred. It is not allowed to carry on looking for address records in this circumstance.

Controlling DNS lookups

There are two options that control the way DNS lookups are done. The resolver option `RES_DEFNAMES` is set by default. This causes the resolver to qualify domains that consist of just a single component (that is, contain no dots) with a default domain, which is normally the name of the local host minus its leading component. So, for example, on a host called *dictionary.book.example*, the effect of looking up the domain *thesaurus* would be to look for *thesaurus.book.example*. This is usually a useful behavior for groups of hosts in the same superior domain, which is why it happens by default. However, it can be disabled if necessary by setting `no_qualify_single`.

The resolver option `RES_DNSRCH` is not set by default, but can be requested by setting `search_parents`. In this case, if the initial lookup fails, the resolver searches the default domain and its parent domains. Continuing the previous example, the effect of looking up *animal.farm* with this option is first to look it up as given, and if that fails, to look up *animal.farm.book.example*. The option is turned off by default because it causes problems in domains that have wildcard MX records.* Suppose the following record:

```
*.example.  MX  6  mail.example.
```

exists and there is a host called *a.book.example* that has no MX records of its own. What happens when a user on some other host in the *book.example* domain mails to *someone@a.book*? On failing to find an MX record for *a.book*, if `search_parents` is set, the resolver goes on to try *a.book.book.example*, which matches the wildcard MX record but is likely to be totally inappropriate.

Conditions for MX Records

Two other **lookuphost** options affect what is done after MX records have been looked up in the DNS. If `check_secondary_mx` is set, the router declines unless the local host is found in the list of hosts obtained from an MX lookup.† This identifies domains for which the local host is an MX backup, and can therefore be used to process these domains in some special way. It differs from the generic `self` option, which applies only when the *lowest* numbered MX record points to the local host.

One of the problems of the proliferation of personal computers on the Internet is that very many of them do not run MTAs, yet if their DNS-registered domain names appear in email addresses, sending MTAs are obliged to try to deliver to them using their DNS address records. A sending MTA normally tries for several days before giving up. This can easily happen if an MUA on a workstation is incorrectly configured so that it sends out mail containing its own domain name in return addresses, instead of using the domain that refers to its email server.

The RFCs still mandate the use of address records when MX records do not exist, and there are still hosts on the Internet that rely on this behavior. In general, therefore, you cannot do anything about this problem. However, if you know that there are MX records for all your own email domains, you can avoid the problem within your own local network, by setting, for example:

```
mx_domains = *.your.domain
```

on the **lookuphost** router. For any domain that matches `mx_domains`, Exim looks only for MX records. It does not go on to look for address records when there are no MX records. This means that domains without MX records are immediately bounced instead of being retried.

* Wildcard MXs are useful mostly for domains at non-IP-connected sites. Because their effects are often not what is really wanted, they are rarely encountered.

† The local host, and any with greater or equal MX preference values, are then removed from the list, in accordance with the usual MX processing rules.

Using the System's Host lookup Instead of the DNS

If you set the `gethostbyname` option for a **lookuphost** router, it does not call the DNS resolver, and the settings of `qualify_single`, `search_parents`, `check_secondary_mx`, and `mx_domains` are ignored. Instead of doing a DNS lookup, the system's host lookup function is called to find an IP address for the host whose name is the same as the domain in the email address.* Thus, there is no domain indirection as is the case when MX records are used; the email domain and the hostname are one and the same. For example:

```
namedhosts:
  driver = lookuphost
  domains = *.mydom.example
  transport = remote_smtp
  gethostbyname
```

This could be useful for hosts on a LAN that is not connected to the Internet, though the **domainlist** router provides another way of doing the same thing. The action taken depends on how host name lookup is configured on your operating system.† Commonly, it searches */etc/hosts* first, and may then go on to do a NIS and/or a DNS lookup if necessary. If the DNS is used, however, no MX processing is performed; only address records are looked up. Some of these lookup methods support hostname abbreviation; for example, a line in */etc/hosts* could be:

```
192.168.131.111    yellow.csi.example.com    yellow
```

where *yellow* is an abbreviation for the full name *yellow.csi.example.com*.

Explicit lookup Widening

When either kind of host lookup fails, **lookuphost** can be configured to try adding specific strings onto the end of the domain name, using the `widen_domains` option. This provides a more controlled extension mechanism than `search_parents` does for DNS lookups, because instead of searching every enclosing domain, just those extensions that you specify are used. Furthermore, in the case of DNS lookups, it operates only after both an MX and an address record lookup have failed, thereby avoiding the problem with the wildcard MX records that were previously mentioned. For example, suppose we have the following:

```
widen_domains = cam.ac.example : ac.example
```

on a host called *users.mail.cam.ac.example*, and a user on that host sends mail to the domain *semreh.cam*. First, **lookuphost** looks for MX and address records for

* The IPv4 function is `gethostbyname()`, which gives its name to this option. In IPv6 systems, `getipnodebyname()` (or, on some systems, `gethostbyname2()`) is used.

† The file */etc/nsswitch.conf* is used on several Unix variants, including Linux and Solaris, to configure what `gethostbyname()` does.

semreh.cam, and because `search_parents` is not set, the resolver does no widening of its own. As there is no top-level domain called *cam*, the lookup fails, so **lookuphost** goes on to try *semreh.cam.cam.ac.example* and then *semreh.cam.ac.example* as a result of the setting of `widen_domains`, but no other widening is done.

Header Rewriting

When an abbreviated name is expanded to its full form, either as part of lookup processing, or as a result of the `widen_domains` option, all occurrences of the abbreviated name in the header lines of the message are rewritten with the full name.* This can be suppressed by setting `no_rewrite_headers`, but this option should be turned off only when it is known that no message is ever going to be sent outside an environment where the abbreviation makes sense.

Summary of lookuphost Options

The options that are specific to **lookuphost** are summarized in this section:

`check_secondary_mx` (Boolean, default = false)
: If this option is set, the router declines unless the local host is found in the list of hosts obtained by MX lookup. This can be used to process domains for which the local host is a secondary mail exchanger differently from other domains.

`gethostbyname` (Boolean, default = false)
: If this is true, the `gethostbyname()` function is used to look up the domain name as a hostname, and the options relating to the DNS are ignored. Otherwise, the name is looked up in the DNS, and MX processing is performed.

`mx_domains` (domain list, default = unset)
: This option applies to domains that are looked up directly in the DNS. A domain that matches `mx_domains` is required to have an MX record in order to be recognized.

`qualify_single` (Boolean, default = true)
: If domains are being looked up in the DNS, the resolver option that causes it to qualify single-component names with the default domain (`RES_DEFNAMES`) is set.

* When an MX record is looked up in the DNS and matches a wildcard record, name servers normally return a record containing the name that has been looked up, making it impossible to detect whether a wildcard is present or not. However, some name servers have recently been seen to return the wildcard entry itself. If the name returned by a DNS lookup begins with an asterisk, Exim does not use it for header rewriting.

`rewrite_headers` (Boolean, default = true)
An abbreviated name may be expanded to its full form when it is looked up, or as a result of the `widen_domains` option. If this option is true, all occurrences of the abbreviated name in the headers of the message are rewritten with the full name.

`search_parents` (Boolean, default = false)
If domains are being looked up in the DNS, the resolver option that causes it to search parent domains (`RES_DNSRCH`) is set if this option is true. This is different from the `qualify_single` option in that it applies to domains containing dots.

`widen_domains` (string list, default = unset)
If a lookup fails and this option is set, each of its strings in turn is added onto the end of the domain and the lookup is tried again. This option applies to lookups using `gethostbyname()` as well as to DNS lookups. Note that when the DNS is being used for lookups, the `qualify_single` and `search_parents` options cause some widening to be undertaken inside the DNS resolver.

The domainlist Router

Sometimes, you know exactly how you want to route a particular domain. For instance, a client host on a dial-up connection normally sends all outgoing mail to a single host (often called a *smart host*) for onward transmission. A less trivial example is the case of a gateway that is handling all incoming mail to a local network. In this case, you will know that certain domains should be routed to specific hosts on your network. The MX records for these domains will point to the gateway, so another means of routing onwards from the gateway is required.

The **domainlist** router exists in order to handle this kind of manual routing. It is so called because it is configured with a list of domains that it is to handle, together with information as to what to do with them. This is the router that is most commonly used for defining explicit routing requirements (that is, for manually routing certain remote domains).

When **domainlist** matches a domain, one of the following actions can be specified:

- The address is queued for a remote transport, along with a list of hosts taken from the configuration. This is probably the most common way of configuring **domainlist**. It is Exim's way of specifying the routing instruction "Send mail for this domain to one of these hosts." The remote transport tries to deliver the message to each host in turn, until one accepts it. If `hosts_randomize` is set, the list of hosts in the configuration is sorted randomly each time it used; otherwise, the order is preserved. The hosts' IP addresses are obtained either by calling the system host lookup function, or by doing a DNS lookup, with or without MX processing.* If the first host in the list turns out to be the local host, the generic `self` option controls what happens.

- The address is passed on to the next router with a new domain name. This can be thought of as "route mail for this domain as if it were that domain." This action does not change the domain name in the message or its envelope, but it causes the message to be routed by the subsequent routers according to the new name.
- The address is queued for a local transport, with a hostname optionally passed as data. This can be used to store messages for remote hosts in local files, and is often used as a way of storing mail for dial-up clients.

We will give some examples of all three types as we describe how the routing rules are set up. There are in fact two ways of doing this: the rules can be given inline in the configuration file, or they can be stored in a file and looked up on a per-domain basis.

Inline Routing Rules

Inline rules are given by the `route_list` option. Each rule has up to three parts, separated by whitespace:

1. A pattern that matches the domains to be handled by the rule.
2. A host list. This is an expanded item, which means that it might contain internal whitespace. If this is the case, you must enclose the host list in quotes.
3. An option that specifies how the IP addresses for the hosts are looked up.

The domain pattern is the only mandatory item in the rule. (You don't, for example, need a host list if the router is sending addresses to a local transport.) The pattern is in the same format as one item in a domain list such as `local_domains`. For example, it may be wildcarded, a regular expression, or a file or database lookup.†

If the pattern at the start of a rule is a lookup item, the data that was looked up is available in the variable $value during the expansion of the host list. For example:

```
route_list = dbm;/etc/rdomains $value:backup.example byname
```

matches domains by a DBM lookup; the data from the lookup is used in the host list, with an additional backup hostname.

* If MX processing is specified, the list is strictly a domain list rather than a host list.

† See the section "Domain Lists," in Chapter 18, *Domain, Host, and Address Lists*, for a full description of the available formats.

If the pattern at the start of a rule in `route_list` is a regular expression, the numeric variables $1, $2, and so on hold any captured substrings during the expansion of the host list. A setting such as:

```
route_list = ^(ab\d\d)\.example  $1.mail.example  byname
```

routes mail for the domain *ab01.example* (for example) to the host *ab01.mail.example*.

The simplest use of **domainlist** is found on client hosts that send all nonlocal addresses to a single smart host for onward delivery. Such a configuration has just a single router:

```
smarthost:
  driver = domainlist
  transport = remote_smtp
  route_list = *  smarthost.example.com  bydns_a
```

A single asterisk as a domain pattern matches all domains, so this router causes all messages containing remote addresses to be sent to the host *smarthost.example.com*.

The IP address for the host is (in this example) obtained from its DNS address record as a result of the option `bydns_a`. The available options for specifying the address lookup are as follows:

`byname`
: Find an address using the system's host lookup function instead of doing a DNS lookup directly. This option is also used if an explicit IP address is given instead of a hostname.

`bydns_a`
: Do a DNS lookup for address records. That is, do *not* look for MX records.

`bydns`
: Do full DNS processing on the name, looking for MX records or address records (thus treating the name as a new mail domain name rather than strictly as a hostname).

`bydns_mx`
: Do a DNS lookup, but insists on the existence of an MX record (so again, this is treating the name as a mail domain, not a hostname).

If a host is found not to exist, delivery is deferred and the message is frozen, on the grounds that this is most probably a configuration error. However, a different action can be requested by setting the `host_find_failed` option. The values it can take are as follows:*

`freeze`
  Delivery is deferred, and the message is frozen. This is the default action.

`defer`
  Delivery is deferred, but the message is not frozen.

`pass`
  The router declines, and so the address is offered to the next router. This setting overrides `no_more`.

`fail`
  The router declines, but no more routers are tried, so the address fails.

So far we have been discussing a single inline routing rule, but `route_list` can contain any number of such rules. Exim scans them in order until it finds one that matches the domain of the address it is handling. If none of them match, the router declines. When there is more than one rule, they are separated by semicolons, because a colon is used as the separator in the host lists. Here is an example that uses a setting of `route_list` containing several rules:

```
private_routes:
  driver = domainlist
  transport = remote_smtp
  route_list = domain1.example    host1.example  byname; \
               *.domain2.example  host2.example:host3.example  bydns_a; \
               domain3.example    192.168.45.56  byname
```

This router operates as follows:

- The domain *domain1.example* is routed to the host *host1.example*, whose IP address is looked up using the system host lookup function.
- The domains that match **.domain2.example* are routed to the two hosts *host2.example* and *host3.example*, whose IP addresses are obtained from the DNS address records.
- The domain *domain3.example* is routed to the host whose IP address is 192.168.45.56. The use of `byname` is required; if you don't specify a `lookup` option, Exim thinks you are trying to pass a new domain name to the next router instead of sorting out the IP addresses here.
- All other domains are passed on to the next router.

A **domainlist** router is, of course, also subject to any setting of the generic `domains` option.

* In earlier versions of Exim, `fail_soft` and `fail_hard` were used instead of `pass` and `fail`. The older settings are still recognized.

Looked-up Routing Rules

Sometimes is it more convenient to keep routing information in a file or database rather than include it inline in the configuration file. If you want to do this, you need to set `search_type` to specify the kind of lookup (for example, `lsearch` or `ldap`). You can, if you like, set `route_list` as well as specifying a lookup. In this case, `route_list` is searched first, and the lokup is performed only if none of its rules match the domain. If the lookup fails to find any routing data, the router declines.

If `search_type` is one of the single-key lookup types, `route_file` must be set to the name of the lookup file. For example:

```
private_routes:
  driver = domainlist
  transport = remote_smtp
  search_type = cdb
  route_file = /etc/routes.cdb
```

This specifies that the routing data is to be obtained by looking up up the domain in */etc/routes.cdb* using a *cdb* lookup. Partial single-key lookups (see the section "Partial Matching in Single-Key Lookups," in Chapter 16, *File and Database Lookups*) may be used to cause a set of domains all to use the same routing data.

For a query-style lookup type, a single query can be given in `route_query`, or a colon-separated list of queries can be given in `route_queries`. For example:

```
private_routes:
  driver = domainlist
  transport = remote_smtp
  search_type = pgsql
  route_query = select routedata from routelist where domain='$domain';
```

The routing data returned from a successful lookup must be a string containing a host list and options, separated by whitespace. These are used in exactly the same way as described earlier for inline routing rules. The final example in the previous section, if reorganized to use a file lookup, would be configured like this:

```
private_routes:
  driver = domainlist
  transport = remote_smtp
  search_type = partial-lsearch*
  route_file = /etc/routes
```

with the file containing:

```
domain1.example:    host1.example  byname
*.domain2.example:  host2.example:host3.example  bydns_a
domain3.example:    192.168.45.56  byname
```

When the rule is found by a partial single-key lookup in `route_file`, $1 contains the wild portion of the domain name during the expansion of the host list.

As neither the host list nor the options are compulsory in all circumstances, the data returned from a lookup can legitimately be an empty string in some cases.

If the domain does not match anything in `route_list`, and looking it up using `route_file`, `route_query` or `route_queries` also fails, the router declines to handle the address, so it is offered to the next router (unless `no_more` is set).

Preprocessing the Host List

If a host list is present in the rule, it is expanded before use, and the result of the expansion must be a colon-separated list of names. Literal IP addresses may also appear, but only when the option is set to `byname`. Some string expansion items may contain whitespace, and if this is the case, the host list must be enclosed in single or double quotes to avoid premature termination. For example:

```
myroutes:
  driver = domainlist
  transport = remote_smtp
  route_list = *.mydoms.example \
    "${lookup {$domain} lsearch {/etc/routes}{$value}fail}" byname
```

The string expansion in this configuration uses the domain name as a key for a lookup, in order to obtain a host list from the */etc/routes* file. If the lookup fails, the expansion is forced to fail, causing the router to decline. Unless `no_more` is set, the address is offered to the next router. If expansion fails for some other reason, the message is frozen, because this is considered to be a configuration error.

During the expansion of the host list, $0 is always set to the entire domain that is being routed. This may differ from $domain, which contains the original domain of the address. If the address has been processed by a previous **domainlist** router that passed on a different routing domain, $0 contains this new routing domain, whereas $domain contains the original domain.

Routing to a Local Transport

Routers are not constrained to using remote transports. They can also arrange for addresses to be passed to local transports. An instance of **domainlist** that is configured in this way is often used for handling messages for dial-up hosts. Rather than leaving them on Exim's queue, where they will be uselessly retried, they are delivered into files from which they can be retrieved when the client host connects.

If there is no host list in a routing rule (and therefore necessarily no options either), a local transport (that is, not an SMTP transport) must be specified for the router via the generic `transport` option. The address is routed to the transport, and the route list entry can be as simple as a single domain name in a configuration such as this:

```
route_append:
  driver = domainlist
  transport = batchsmtp_appendfile
  route_list = gated.domain.example
```

This router causes the **batchsmtp_appendfile** transport to be run for addresses in the domain *gated.domain.example*. Normally, instead of just a single domain, some kind of pattern would be used to match a set of domains, and a user needs to be specified for running the transport. A complete configuration might contain a transport such as this:

```
dialup_transport:
  driver = appendfile
  bsmtp = domain
  file = /var/dialups/$domain
  user = exim
```

and a router such as this:

```
route_dialup:
  driver = domainlist
  transport = dialup_transport
  route_list = *.dialup.example.com
```

The setting of `bsmtp` in the transport requests that the message's envelope be preserved in *batch SMTP* (BSMTP) format. We cover how this works and what it can be used for in the section "Batched Delivery and BSMTP," in Chapter 9, *The Transports*.

When a local transport is used like this, a single hostname may optionally be present in each routing rule. It is passed to the transport in the variable $host, and could, for example, be used in constructing the filename. The following router matches two sets of domains; for each set, a different value appears in $host.

```
route_list = *.dialup1.example.com host1; \
             *.dialup2.example.com host2
```

You can use **domainlist** for routing mail directly to UUCP software. Here is an example of one way this can be done, taken from a real configuration. A **pipe** transport is used to run the UUCP software directly (see the section "The pipe Transport," in Chapter 9):

```
uucp:
  driver = pipe
  user = nobody
  command = /usr/local/bin/uux -r - $host!rmail $local_part
  return_fail_output = true
```

This transport substitutes the values of $host and $local_part in the command:

```
/usr/local/bin/uux -r - $host!rmail $local_part
```

and then runs it as the user *nobody*. The message is piped to the command on its standard input. If the command fails, any output it produces is returned to the sender of the message. The corresponding router is:

```
uucphost:
  transport = uucp
  driver = domainlist
  route_file = /usr/local/exim/uucphosts
  search_type = lsearch
```

The file */usr/local/exim/uucphosts* contains entries such as this:

```
darksite.ethereal.example:  darksite
```

When the router processes the address *someone@darksite.ethereal.example*, it passes the address to the **uucp** transport, setting $host to the string `darksite`.

Using domainlist on a Mail Hub

A *mail hub* is a host that receives mail for a number of domains (usually, but not necessarily, via MX records in the DNS), and delivers it using its own private routing mechanism. Often the final destinations are behind a firewall, with the mail hub being the one machine that can connect to machines both inside and outside the firewall. The **domainlist** router on the hub can be set to handle incoming mail like this:

```
through_firewall:
  driver = domainlist
  transport = remote_smtp
  route_file = /internal/host/routes
  search_type = lsearch
```

If there are only a small number of domains, the routing could be specified inline, using the `route_list` option, but for a larger number, a lookup is easier to manage. If a routing file itself becomes large (more than, say, 20 to 30 entries), it is a good idea to turn it into one of the indexed formats (DBM or cdb) to improve performance. For this example, the file containing the internal routing might contain lines such as this:

```
abc.ref.example: m1.ref.example:m2.ref.example byname
```

The DNS would be set up with an MX record for *abc.ref.example* pointing to the mail hub, which would forward mail for this domain to one of the two specified hosts and look up their addresses using `gethostbyname()`. They would be tried in order, because `hosts_randomize` is not set.

If the domain names are, in fact, the names of the machines to which the mail is to be sent by the mail hub, the configuration can be simplified. For example:

```
hub_route:
  driver = domainlist
  transport = remote_smtp
  route_list = *.rhodes.example  $domain  byname
```

This configuration routes domains that end in *.rhodes.example* by calling `gethostbyname()` on the domain.* A similar approach can be taken if the hostname can be obtained from the domain name by any of the transformations that are available in Exim's string expansion mechanism.

Varying the Transport

In addition to specifying how names are to be looked up, the options part of a rule may also contain a transport name. This is then used for domains that match the rule, overriding any setting of the generic `transport` option. For example, this router uses different local transports for each of its rules:

```
route_append:
  driver = domainlist
  route_list = \
    *.gated.domain1  $domain  batch_appendfile; \
    *.gated.domain2  ${lookup{$domain}dbm{/etc/domain2/hosts}\
                       {$value}fail}  batch_pipe
```

The first rule sends the address that it matches to the **batch_appendfile** transport, passing the domain in the $host variable, which does not achieve much (since it is also in $domain); the second rule does a file lookup to find a value to pass in $host to the **batch_pipe** transport, specifying that the router should decline to handle the address if the lookup fails.

Using domainlist to Change the Routing Domain

The examples used so far have all routed to a specific remote transport. This is the most common use for **domainlist**. It provides a way of implementing local rules for routing certain domains to specified hosts. However, **domainlist** can be used without a transport.

If no transport is specified, the **domainlist** router does not set up a delivery for the domains it matches. Instead it replaces the domain name that is being routed by a new domain taken from the host list, and passes that domain to the following router. It is a configuration error to omit a transport without specifying a new domain name.

* This particular routing could equally well have been done using a **lookuphost** router with the `gethostbyname` option. There is often more than one way to do something in Exim.

Other domainlist Options

The remaining options for **domainlist** fall into in two groups. The options `modemask`, `owners`, and `owngroups` control the mode and ownership of the file specified in `route_file`, when it is a real file. These options act in exactly the same way as they do for the **aliasfile** and **forwardfile** directors (see the section "Options Common to aliasfile and forwardfile," in Chapter 7, *The Directors*). The `no_qualify_single` and `search_parents` options apply to any DNS lookups that are done, and act in the same way as they do in the **lookuphost** router (see the section "The lookuphost Router," earlier in this chapter).

Summary of domainlist Options

The options that are specific to **domainlist** are summarized in this section:

`host_find_failed` (string, default = `freeze`)
: This option controls what happens when a host that **domainlist** tries to look up (because an address has been specifically routed to it) does not exist. The option can be set to one of the following:

```
freeze
defer
pass
fail
```

The default assumes that this state is a serious configuration error. The difference between `pass` and `fail` is that the former causes the address to be passed to the next router, overriding `no_more`, while the latter does not, causing the address to fail completely. This option applies only to a definite "does not exist" state; if a host lookup suffers a temporary error, delivery is deferred unless the generic `pass_on_timeout` option is set.

`hosts_randomize` (Boolean, default = false)
: If `hosts_randomize` is false, the order in which hosts are listed is preserved as an order of preference for delivering the message; if it is true, the list is shuffled into a random order each time it is used.

`modemask` (octal integer, default = 022)
: This specifies mode bits that must not be set for the route file. If they are set, delivery is deferred and the message is frozen.

`owners` (string list, default = unset)
: This specifies a list of permitted owners for the route file. If it is unset, no check on the ownership is done. If the file is not owned by a user in the list, delivery is deferred and the message is frozen.

`owngroups` (string list, default = unset)
: This specifies a list of permitted group owners for the route file. If it is unset, no check on the file's group is done. If the file's group is not in the list, delivery is deferred and the message is frozen.

`qualify_single` (Boolean, default = true)
: For any domain that is looked up in the DNS, the resolver option that causes it to qualify single-component names with the default domain (`RES_DEFNAMES`) is set if `qualify_single` is true.

`route_file` (string, default = unset)
: If this option is set, `search_type` must be set to one of the single-key lookup types, and `route_query` must not be set. The domain being routed is used as the key for the lookup, and the resulting data must be a routing rule. The filename is expanded before use.

`route_list` (string list, default = unset)
: This string is a list of routing rules. Note that, unlike most string lists, the items are separated by semicolons by default. This is so that they may contain colon-separated host lists.

`route_queries` (string, default = unset)
: This option is an alternative to `route_query`; the two options are mutually exclusive. The difference is that `route_queries` contains a colon-separated list of queries, which are tried in order until one succeeds or defers, or all fail. Any colon characters actually required in an individual query must be doubled, in order that they not be treated as query separators.

`route_query` (string, default = unset)
: If this option is set, `search_type` must be set to a query-style lookup type, and `route_file` must not be set. The query is expanded before use, and during the expansion, the variable $domain contains the domain being routed. The data returned from the lookup must be a routing rule.

`search_parents` (Boolean, default = false)
: For any domain that is looked up in the DNS, the resolver option that causes it to search parent domains (`RES_DNSRCH`) is set if `search_parents` is true. This is different from the `qualify_single` option in that it applies to domains containing dots.

`search_type` (string, default = unset)
: This option is mandatory when `route_file`, `route_query`, or `route_queries` is specified. It must be set to one of the supported search types (for example, `lsearch`). Partial single-key lookups can be used.

The ipliteral Router

In the early days of the Internet, before the general availability of the DNS, domain names were not always available. For this reason, RFCs 821 and 822 allow for the use of an *IP domain literal* instead of a domain name in an email address. For example:

```
A.User@[192.168.3.4]
```

The intention is that such an address causes the message to be delivered to the host with that IP address. These days, allowing end users to direct messages to individual machines is not something many administrators are prepared to permit, and the increasing use of firewalls often makes it impossible. However, since the facility is still current in the RFCs, Exim supports it, though the options that set it up are commented out in the default configuration file.

The **ipliteral** router has no specific options of its own. It simply checks whether the domain part of an address has the format of a domain literal, and if so, routes the address to a transport specified by the generic `transport` option, passing the IP address to which the message should be sent. If a domain literal turns out to refer to the local host, the generic `self` option determines what happens.

The queryprogram Router

The **queryprogram** router routes an address by running an external command and acting on its output. The command's job is to make decisions about the routing of the address. It is not a command to perform message delivery; that function is available via the **pipe** transport.

Running an external command is an expensive way to route, and is intended mainly for use in lightly loaded systems or for performing experiments. However, if it is possible to restrict this router to a few lightly used addresses by means of the `domains`, `local_parts`, or `condition` generic options, it could sensibly be used in special cases, even on busy systems.

The queryprogram Command

The command that is to be run is specified in the `command` option. This string is expanded; after expansion it must start with the absolute pathname of the command. An expansion failure causes delivery to be deferred and the message to be frozen. The command is run in a subprocess directly from Exim, without using an intervening shell. If you want a shell, you have to specify it explicitly; because this interposes yet another process, it increases the expense and is not recommended.

Running the queryprogram Command

The command is run in a separate process under a uid and gid that are specified by `command_user` and `command_group`. If the latter is not set, the gid associated with the user is used, as in this example:

```
pgm_router:
  driver = queryprogram
  transport = remote_smtp
  command = /usr/exim/pgmrouter $local_part $domain
  command_user = mail
```

If neither `command_user` nor `command_group` are set, the command is run as the user defined by the `nobody_user` and `nobody_group` options; if they are not set, the command is run as the user *nobody*, if such a user exists. If no user can be found under which to run the process, delivery is deferred and the message is frozen. The directory that is made current while running the command is specified by `current_directory`. It defaults to the root directory.

The Result of the queryprogram Command

The **queryprogram** router has a `timeout` option, which defaults to `1h` (1 hour). If the command does not complete in this time, its process group is killed, delivery is deferred, and the message is frozen. A zero time specifies no time limit, but this is not recommended. The message is also frozen if the command terminates in error (that is, if its return code is not zero.

No input is provided for the command, so any data it requires must be passed in as arguments. The standard output of the command is read when the command terminates successfully. It should consist of a single line of output, containing up to five fields separated by whitespace.

The first field is one of the following words:

`OK`
: Routing did not succeed; offer the address to the next router unless `no_more` is set.*

`FORCEFAIL`
: Routing failed; do not pass the address to any more routers. This means that the address fails.

* In earlier versions of Exim, `FAIL` was used instead of `DECLINE`. It is still recognized, for backwards compatibility.

`DEFER`
: Routing could not be completed at this time; try again later.

`ERROR`
: Some disastrous error occurred; freeze the message.

When the first word is not `OK`, the remainder of the line is an error message explaining what went wrong. For example:

```
DECLINE  cannot route to unseen.discworld.example
```

When the first word is `OK`, the line must be formatted as follows:

```
OK transport-name new-domain option arbitrary-text
```

The second field is the name of a transport instance, or a plus character, which means that the transport specified for the router using the generic `transport` option is to be used.

If the *`new-domain`* field is present, it contains a new domain name that replaces the current one. When, in addition, a transport is specified (either in the second field or by the `transport` option) and the *`option`* field is present, it specifies the method of looking up the new domain name. This can be one of the words `byname`, `bydns`, `bydns_a`, or `bydns_mx`. These operate in exactly the same way as in the **domainlist** router, described earlier in this chapter. For example, the following:

```
OK  remote_smtp  gate.star.example  bydns_a
```

causes the message to be sent using the **remote_smtp** transport to the host *gate.star.example*, whose IP address is looked up using DNS address records. If the host turns out to be the local host, what happens is controlled by the generic `self` option.

The final (fifth) field, if present, is made available to the transport via the variable $route_option. If you want to set the fifth field without specifying a hostname or a lookup method, you must use plus characters as placeholders for the missing fields. For example, a line such as:

```
OK special + + /computed/filename
```

sends the message to the **special** transport, which can use $route_option in its configuration to access the text `/computed/filename`.

If no transport is specified in the response line (that is, a plus character is given) and the generic `transport` option is also unset, the fourth and fifth fields are ignored and the (possibly changed) domain is passed to the next router. This provides a way of dynamically changing the domain that is being routed. That is, it allows the routing program to implement the rule "route this domain as if it were that domain."

Summary of queryprogram Options

The options specific to **queryprogram** are summarized in this section:

`command` (string, default = unset)
This option must be set, and must, after expansion, start with a slash character. It specifies the command that is to be run. Failure to expand causes delivery to be deferred and the message to be frozen.

`command_group` (string, default = unset)
This option specifies a gid to be set when running the command. If it begins with a digit, it is interpreted as the numerical value of the gid. Otherwise it is looked up in the password data.

`command_user` (string, default = unset)
This option specifies the uid, which is set when running the command. If it begins with a digit, it is interpreted as the numerical value of the uid. Otherwise, the corresponding uid is looked up in the password data, and if `command_group` is not set, a value for the gid is taken from the same entry.

`current_directory` (string, default = unset)
This option specifies an absolute path, which is made the current directory before running the command. If it is not set, the root directory is used.

`timeout` (time, default = `1h`)
If the command does not complete within the timeout period, its process group is killed and the message is frozen. A value of zero time specifies no timeout.

9 The Transports

Transports are the modules within Exim that carry out the actual deliveries. Some examples of transports have appeared in earlier chapters; in this chapter, we talk about the options that apply to all of them, and then consider each particular transport in turn. The available transports are as follows:

appendfile
: A transport that writes messages to local files.

autoreply
: A transport that generates automatic replies to messages.

lmtp
: A transport that delivers messages to external processes using the LMTP protocol.

pipe
: A transport that passes messages to external processes via pipes.

smtp
: A transport that writes messages to other hosts over TCP/IP connections, using either SMTP or LMTP.

autoreply
: Really a pseudotransport, because it does not actually deliver the message anywhere; instead it generates a new outgoing message (an automatic reply). It is included among the transports because its method of operation and configuration are the same, and it can include a copy of the original message in the reply.

Options Common to All Transports

There are several generic options that can be set for any transport, though some of them are used almost exclusively on local transports. The only required option is `driver`, which defines which transport is being configured. For the **smtp** transport, this is often the only option you need to provide. In Exim's default configuration file, it is configured as follows:

```
remote_smtp:
  driver = smtp
```

In this case, the host to which the message is to be transported is expected to be supplied by a router (for example, as a result of a DNS lookup in **lookuphost**), and all the other transport options are defaulted.

Debugging Transports

The `debug_print` option operates just like the router and director option of the same name. Its sole purpose is to help debug Exim configurations. When Exim is run with debugging turned on,* the string value of `debug_print` is expanded and added to the debugging output when the transport is run. This facility can be used to check that the values of certain variables are what you think they should be. For example, with the following:

```
remote_smtp:
  driver = smtp
  debug_print = self_hostname = $self_hostname
```

the value of the variable $self_hostname would be added to the debugging output at the time the transport was run.

Transporting Only Part of a Message

Normally, the job of a transport is to copy an entire message. For special purposes, however, it is possible to transport only the header lines or only the body. Setting `headers_only` or `body_only`, respectively, achieves this, but only one of them may be set at once. For example, if copies of messages are being taken for some kind of header analysis, `headers_only` reduces the amount of data that is written.

* See *-d* and *-v* in the section "Options for Debugging," in Chapter 20, *Command-Line Interface to Exim*.

Controlling Message Size

A transport can be configured to reject messages above a certain size by setting `message_size_limit` to a value greater than zero. The default is to apply no limit. Deliveries of messages that are above the limit fail and the address is bounced. If there is any chance that the bounce message (which contains a copy of the original message) could be routed to the same transport, you should ensure that the configuration option `return_size_limit` is less than the transport's `message_size_limit`, as otherwise the bounce message will also fail.

Note that there is a main configuration option for limiting the size of all messages processed by Exim, whose name is also `message_size_limit`. To have any effect, a local setting on a transport must naturally be less than the global limit.

Adding and Removing Header Lines

During the delivery process, a transport can be configured to add or remove header lines. Three specific headers are commonly required to be added when a message is delivered into a local mailbox, and separate options are provided to request them. They are added at the very start of the message, before the *Received:* header lines, and are as follows:

`delivery_date_add`

Requests the addition of a header line of the form:

```
Delivery-date: Tue, 29 Feb 2000 16:14:32 +0000
```

which records the date and time that the message was delivered.

`envelope_to_add`

Requests the addition of a header line of the form:

```
Envelope-to: alex@troy.example
```

which records the original envelope recipient address that caused the delivery to occur. This may be an address that does not appear in the *To:* or *Cc:* header lines. In cases where a single delivery is being done for several recipient addresses, there may be more than one address listed.

`return_path_add`

Requests the addition of a header line of the form:

```
Return-path: <phil@thesa.example>
```

which records the sender address from the message's envelope. For a bounce message, the added header is:

```
Return-path: <>
```

RFC 822 states that the *Return-path:* header "is added by the final transport system that delivers the message to its recipient." It should not, therefore, be present in incoming messages. The other two added headers are not standardized (though they are used by some other MTAs) and also should not be present in incoming messages. Exim therefore removes all three of these headers from messages it receives, to allow messages that have already been delivered to be easily resent.*

Other header lines can be added to the message by means of the `headers_add` option. If this is set, its contents are added at the end of the header section at the time the message is transported. Header lines added by a transport follow any that are added to an address by directors or routers. Multiple lines can be added by coding `\n` inside quotes. For example:

```
headers_add = "\
  X-added: this is a header added at $tod_log\n\
  X-added: this is another"
```

Exim does not check the syntax of these added header lines; you should ensure that they conform to RFC 822. A newline is supplied at the end if one is not present.

Original header lines (those that were received with the message) can be removed by setting `headers_remove` to a list of their names for example:

```
headers_remove = return-receipt-to : acknowledge-to
```

The header names are given without their terminating colons (the colon in the example is a list separator character). These header lines are removed before the addition of any new ones specified by `headers_add`, so an individual header line can be removed and replaced by something different. However, it is not possible to refer to the old contents when defining the new line.

Both `headers_add` and `headers_remove` are expanded before use. If the result is the empty string or the expansion is forced to fail, no action is taken. Other kinds of failure (for example, an expansion syntax error) cause delivery to be deferred.

Header line additions and removals can also be specified on director and router configurations (see the section "Adding or Removing Header Lines," in Chapter 6, *Options Common to Directors and Routers*), in which case they are associated with the addresses that those drivers handle. If one director or router is associated with just one transport, it doesn't matter whether you specify header line changes on the transport or on the router or director. However, in configurations where several routers or directors are using the same transport, where you specify header line changes obviously makes a difference.

* This behavior can be prevented by setting the configuration options `no_return_path_remove`, `no_delivery_date_remove`, and `no_envelope_to_remove`, respectively, but you should not normally do this.

At transport time, the removal list from the address is merged with the removal list from the transport before the relevant header lines are removed. Then the additions from the address and from the transport are done. It is, therefore, not possible to use the transport's option to remove header lines added by a router or director.

Rewriting Addresses in Header Lines

Some installations make a distinction between private email addresses that are used within a single host or within a local network, and public addresses that are used on the Internet. When a message passes from their private network to the outside world, they want internal addresses to be translated into external ones.

We haven't yet talked about Exim's address rewriting facilities. They are described in Chapter 14, *Rewriting Addresses*. Their main focus is on rewriting addresses at the time of a message's arrival, which means that it affects every copy of a message that is delivered. This is no good for solving the problem of internal and external addresses, because a single message may have both internal and external recipients. Some copies may need to be rewritten, whereas others may not.

To overcome this difficulty, a new generic transport option called `headers_rewrite` was added to Exim for Release 3.20. It allows addresses in header lines to be rewritten at transport time (that is, as the message is being copied to its destination). This means that the rewriting affects only those copies of the message that pass through the transport where the option is set. Copies that are delivered by other transports are unaffected.

There's a full discussion of this in Chapter 14, so we won't say any more about it here.

Changing the Return Path

"Return path" is another name for the sender address carried in the message's envelope. If the `return_path` option is set for a transport, its contents are expanded, and the result replaces the existing return path. The expansion can refer to the existing value using the $return_path variable. If the expansion is forced to fail, no replacement occurs; if it fails for another reason, Exim writes a message to its panic log and exits immediately.

The main use of this option is for implementing *variable envelope return paths* (VERP) for messages from mailing lists.* The problem with mailing list deliveries that bounce is that it is often difficult to discover which original recipient address provoked the bounce.

* See *ftp://koobera.math.uic.edu/www/proto/verp.txt.*

Suppose somebody subscribed to a mailing list using the address *J.Smith99@alma.mater.example*, which is forwarded to *jan@plc.co.example*. All goes well until she changes jobs, her email account at *plc.co.example* is cancelled, and she forgets to update the forwarding because there is not much traffic on the list. When next a message is posted, the manager of the list receives a bounce message about a failure to deliver to *jan@plc.co.example*, an address that does not appear on the list. Finding out which original address caused the bounce may be possible by analysis of the *Received:* header lines that were added to the message; these sometimes contain the recipient address in copies of the message that have only one recipient, for example:

```
Received: from [192.168.247.11] (helo=mail.list.example ident=exim)
        by draco.alma.mater.example with esmtp (Exim 3.22 #3)
        id 124h5o-0005wn-00
        for J.Smith99@alma.mater.example; Wed, 20 Jun 2001 10:46:44 +0100
```

However, if the message has more than one recipient, their addresses must not be placed in a *Received:* header because this would constitute a confidentiality exposure.* In any event, diagnosing the problem address is time-consuming, and not something that can easily be automated.

The VERP solution to this problem is to encode the subscriber's address in the envelope sender of the message, so that it is immediately available from any bounce messages. For example, suppose messages to the mailing list were previously sent out with the envelope sender set to:

```
somelist-request@list.example
```

After configuring VERP, the copy of the message that is sent to *J.Smith99@alma.mater.example* has (for example) this envelope sender:

```
somelist-request=J.Smith99%alma.mater.example@list.example
```

If a bounce message is sent back to that address, the address of the subscriber that provoked it can easily be extracted in an automated way.

The downside of VERP is that a separate copy of every message must be sent to each list subscriber, so that it can have a customized sender address. For large lists that may have hundreds of subscribers in the same domain, this can use substantially more bandwidth and take a lot longer in real time. There are two ways to alleviate this problem:

- Use VERP only for an occasional test message, say once a week. This need not be a special message; a normal post to the list could trigger it. Ignore bounces from non-VERP messages.

* Usually, subscribers to a mailing list are not shown the addresses of other subscribers.

- Maintain a list of domains that have many subscribers, and send single copies to those domains. In other words, forgo the benefit of VERP for those domains.

VERP can be supported in Exim by using the `return_path` transport option to rewrite the envelope sender at transport time. For example, the following could be used:

```
return_path = \
  ${if match {$return_path}{^(.+?)-request@list.example\$}\
  {$1-request=$local_part%$domain@list.example}fail}
```

This has the effect of rewriting the return path (envelope sender) if the local part of the original return path ends in `-request` and the domain is *list.example*. The rewriting inserts the local part and domain of the recipient into the return path, in the format used in the previous example.

For this to work, you must arrange for outgoing messages that have `-request` in their return paths to be passed to the transport with just a single recipient, because $local_part and $domain are not set for messages that have multiple recipients. Local transports operate on one recipient at a time by default, but for an **smtp** transport you need to set the following:

```
max_rcpt = 1
```

in the transport's options. If your host doesn't handle much other traffic, you can just set this on the normal **remote_smtp** transport, but if you want to have the benefit of multiple recipients in other cases, you need to set up two **smtp** transports, like this:

```
normal_smtp:
  driver = smtp

verp_smtp:
  driver = smtp
  max_rcpt = 1
  return_path = \
    {${local_part:$return_path}=$local_part%$domain@list.example}fail}
```

and then route mailing list messages to the second of them, using a router such as this:

```
verp_router:
  driver = lookuphost
  transport = verp_smtp
  condition = \
    ${if match {$return_path}{^(.+?)-request@list.example\$}{yes}{no}}
```

The setting of `return_path` on the transport can be simpler, because the strict check is done by the router, so that it sends only addresses that have the `-request` suffix to the transport.

Of course, if you do start sending out messages with this kind of return path, you must also configure Exim to accept the bounce messages that come back to those addresses. Typically this is done by setting a `prefix` or `suffix` option in a suitable director (see the section "Conditional Running of Directors," in Chapter 7, *The Directors*).

The overhead incurred in using VERP depends on the size of the message, the number of recipient addresses that resolve to the same remote host, and the speed of the connection over which the message is being sent. If a lot of addresses resolve to the same host and the connection is slow, sending a separate copy of the message for each address may take substantially longer than sending a single copy with many recipients (for which VERP cannot be used).

Transport Filters

If you want to make more extensive changes than can be achieved with the options just described, or if you want to modify the body of messages as they are transported, you can make use of a *transport filter*. This is a "filter" in the Unix sense of the word; it is unrelated to Exim's message filtering facilities that happen at directing time.

The `transport_filter` option specifies a command that is run at transport time. Instead of copying the message to its destination, Exim uses a pipe to pass it to the command on its standard input. It then reads the standard output of the command and writes that to the destination. This is an expensive thing to do, and is made more so because Exim's delivery process cannot both read from and write to the filtering process, as doing this could lead to a deadlock. It therefore has to create a third process to do the writing, as shown in Figure 9-1.

One possible application of transport filters is to encrypt the bodies of messages as they pass through certain transports. A transport such as:

```
encrypt_smtp:
  driver = smtp
  transport_filter = /usr/mail/encrypt/body $sender_address
```

could be selected by the routers for certain destination addresses, and the value of $sender_address could be used to control how the encryption was done.

The entire message including the header lines, is passed to the filter before any transport-specific processing (such as turning `\n` into `\r\n` and escaping lines starting with a dot for SMTP) is done. The filter can perform any transformations it likes, but, of course, it should take care not to break RFC 822 syntax. A problem might arise if the filter increases the size of a message that is being sent down an

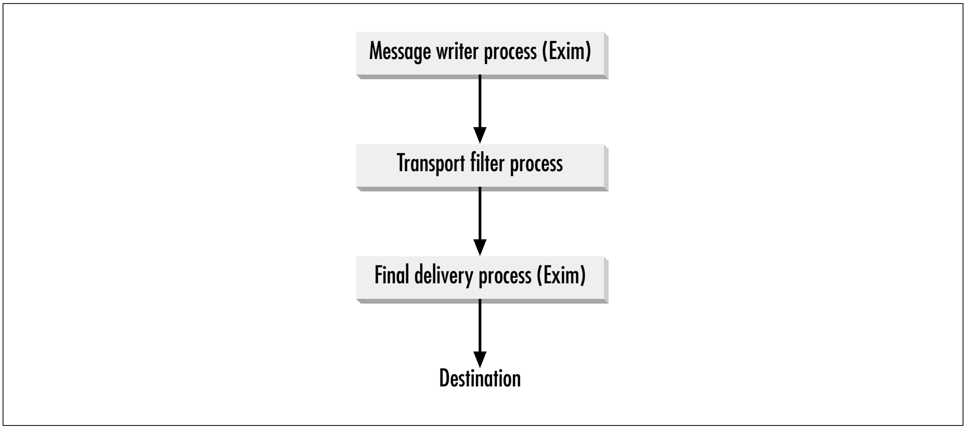


Figure 9-1. Transport filtering

SMTP channel. If the receiving SMTP server has indicated support for the `SIZE` parameter, Exim will have sent the size of the message at the start of the SMTP session. If what is actually sent is substantially more, the server might reject the message. You can work round this by setting the `size_addition` option on the **smtp** transport, either to allow for additions to the message or to disable the use of `SIZE` altogether.

The value of `transport_filter` is the command string for the program that is run in the process started by Exim. This program is run directly, not under a shell. The string is parsed by Exim in the same way as a command string for the **pipe** transport: Exim breaks it up into arguments and expands each argument separately. This means that the expansion cannot accidentally change the number of arguments. The special argument $pipe_addresses is replaced by a number of arguments, one for each address that applies to this delivery.*

The variables $host (containing the name of the remote host) and $host_address (containing the IP address of the remote host) are available when the transport is a remote one. For example:

```
transport_filter = /some/directory/transport-filter.pl \
  $host $host_address $sender_address $pipe_addresses
```

The filter process is run under the same uid and gid as the normal delivery. For remote deliveries, this is the Exim uid and gid.

* $pipe_addresses is not an ideal name for this feature here, but as it was already implemented for the **pipe** transport, it seemed sensible not to change it.

Shadow Transports

A shadow transport is one that is run in addition to the main transport for an address. Shadow transports can be used for a number of different purposes, including keeping more detailed log information than Exim normally provides, and implementing automatic acknowledgment policies based on message headers.

A local transport may set `shadow_transport` to the name of another transport, which must be a local transport that is defined earlier in the configuration file. Shadow remote transports are not supported.

When a shadow transport is defined, and a delivery to the main transport succeeds, the message is also passed to the shadow transport. However, this happens only if `shadow_condition` is unset, or its expansion does not result in a forced expansion failure, the empty string, or one of the strings `0`, `no`, or `false`. This allows you to restrict shadowing to messages that match certain conditions.

If a shadow transport fails to deliver the message, the failure is logged, but it does not affect the subsequent processing of the message. Since the main delivery succeeded, the address is finished with. There is no retrying mechanism for shadow transports.

Only a single level of shadowing is provided; the `shadow_transport` option is ignored on any transport when it is running as a shadow. Options concerned with output from pipes are also ignored. The log line for the successful delivery has an item added on the end, in the following form:

```
ST=<shadow transport name>
```

If the shadow transport did not succeed, the error message is put in parentheses afterwards.

Summary of Generic Transport Options

The options that are common to all the transports are summarized in this section:

`body_only` (Boolean, default = false)
: If this option is set, the message's headers are not transported. The option is mutually exclusive with `headers_only`. If it is used with the **appendfile** or **pipe** transports, the settings of `prefix` and `suffix` should be checked, since this option does not automatically suppress them.

`debug_print` (string, default = unset)
: If this option is set and debugging is enabled, the string is expanded and included in the debugging output when the transport is run.

`delivery_date_add` (Boolean, default = false)
: If this option is true, a *Delivery-date:* header line is added to the message. This gives the actual time the delivery was made.

`driver` (string, default = unset)
: This specifies which of the available transport drivers is to be used. There is no default, and this option must be set for every transport.

`envelope_to_add` (Boolean, default = false)
: If this option is true, an *Envelope-to:* header line is added to the message. This gives the original address in the incoming envelope that caused this delivery to happen. More than one address may be present if `batch` or `bsmtp` is set on transports that support them, or if more than one original address was aliased or forwarded to the same final address.

`headers_add` (string, default = unset)
: This option specifies a string of text that is expanded and added to the header portion of a message as it is transported. If the result of the expansion is an empty string, or if the expansion is forced to fail, no action is taken. Other expansion failures are treated as errors and cause the delivery to be deferred.

`headers_only` (Boolean, default = false)
: If this option is set, the message's body is not transported. It is mutually exclusive with `body_only`.

`headers_remove` (string, default = unset)
: This option is expanded; the result must consist of a colon-separated list of header names (without their terminating colons). Original header lines matching those names are omitted from any message that is transmitted by the transport. However, headers with these names may still be added.

`message_size_limit` (integer, default = 0)
: This option controls the size of messages passing through the transport. If its value is greater than zero and the size of a message exceeds the limit, the delivery fails.

`return_path` (string, default = unset)
: If this option is set, the string is expanded at transport time and replaces the existing return path (envelope sender) value. The expansion can refer to the existing value via $return_path. If the expansion is forced to fail, no replacement occurs; if it fails for another reason, Exim writes to its panic log and exits immediately.

`return_path_add` (Boolean, default = false)
: If this option is true, a *Return-path:* header line is added to the message. This is normally used only on transports that are doing final delivery into a mailbox. If the mailbox is a single file in Berkeley format, the return path is normally available in the separator line, but commonly this is not displayed by MUAs, and so the user does not have easy access to it. Other mailbox formats may not record the return path at all.

`shadow_condition` (string, default = unset)
: See `shadow_transport`.

`shadow_transport` (string, default = unset)
: A local transport may set the `shadow_transport` option to the name of another previously defined local transport. Whenever a delivery to the main transport succeeds, and either `shadow_condition` is unset, or its expansion does not result in a forced expansion failure or the empty string or one of the strings `0` or `no` or `false`, the message is also passed to the shadow transport.

`transport_filter` (string, default = unset)
: This option sets up a filtering process (in the Unix shell sense) for messages at transport time. When the message is about to be written out, the command specified by `transport_filter` is started up in a separate process, and the entire message, including the headers, is passed to it on its standard input. The filter's standard output is read and written to the message's destination.

The smtp Transport

The **smtp** transport is the only remote transport, so it is used for all deliveries to remote hosts. However, more than one instance can be configured with different option settings if necessary. Its most common configuration is very simple, usually this:

```
remote_smtp:
  driver = smtp
```

In this example, all the options that control the parameters of the SMTP connection take their default values. The list of remote hosts must be set up by the router that handled the address, and passed with the address(es) to be delivered.

However, the use of an **smtp** transport is not restricted to routers. It can be used from directors, but these do not set up host lists. To allow for this, the transport itself has options for specifying hosts. In addition, the characteristics of the SMTP connection can be modified in various ways. As a result, there are quite a lot of options for this transport.

Control of Multiple Addresses

The SMTP protocol allows any number of recipient addresses to be passed in a message's envelope, by means of multiple `RCPT` commands.[*] Exim normally does this when a message has more than one address that is routed to the same host, subject to the following options:

- `max_rcpt` specifies the maximum number of `RCPT` commands in one message transfer. The default value is 100. When a message has more than `max_rcpt` recipients going to the same host, an appropriate number of separate copies of the message are sent. If `max_rcpt` is set to 1, a separate copy of the message is sent for each recipient. This is necessary if you want to implement VERP (see the section "Options Common to All Transports," earlier in this chapter).
- When `max_rcpt` is greater than 1, the domains in the addresses need not be the same, provided that they all resolve to the same list of hosts. For example, a set of virtual domains that are all under one management usually all share the same MX hosts. However, if you want to make use of $domain in a transport option, you have to arrange that only one domain is ever involved, because otherwise $domain is not set. You can do this by setting `no_multi_domain`. If you do this, a separate copy of the message is sent for each different domain.

The default action of using multiple addresses in a single transfer is the one recommended by RFC 821. For personal messages (which rarely have more than a couple of recipients) that are traversing well-connected parts of today's Internet, it probably doesn't make much difference; however, for mailing lists with thousands of subscribers, there can be a substantial cost if each is sent a separate copy, especially if many of the addresses are in the same domain.

Control of Outgoing Calls

Because Exim operates in a distributed manner, if several messages for the same host arrive at around the same time, more than one simultaneous connection to the remote host can occur. This is usually not a problem except when there is a slow link between the hosts. In that situation, it may be helpful to restrict Exim to one connection at a time to certain hosts. This can be done by setting `serialize_hosts` to match the relevant remote hosts, for example:

```
serialize_hosts = 192.168.4.5 : my.slow.neighbor.example
```

Exim implements serialization by means of a hints database in which a record is written whenever a process connects to one of the restricted hosts, and deleted

[*] In practice, more than about one hundred recipients should be avoided, as this can lead to problems with some MTAs.

when the connection is ended. Obviously there is scope for records to be left lying around if there is a system or program crash, which would prevent Exim from contacting one of these hosts ever again. To guard against this, Exim ignores any records that are more than six hours old.

If the **smtp** transport finds that the host it is about to connect to has an existing connection, it skips that host and moves on to the next one as if a connection to the host had failed, except that it does not compute any retry information.

If you set up any serialization, you should also arrange to delete the relevant hints databases whenever your system reboots. The names of the files all start with *serialize-`transport-name`*, and they are kept in the *spool/db* directory. There may be one or two files per serialized transport, depending on the type of DBM library in use.

When a message has been successfully delivered over a TCP/IP connection, Exim looks in its hints database to see if there are any other messages awaiting a connection to the same host. If there are, a new delivery process is started for one of them, and the current TCP/IP connection is passed on to it. The new process may in turn create yet another process. Each time this happens, a sequence counter is incremented, and if it ever reaches the value of the `batch_max` option, no further messages are sent on the same TCP/IP connection. However, if `batch_max` is set to zero, no limit is applied. You should not normally need to change this option.

Control of the TCP/IP Connection

When an outgoing SMTP call is made from a host with a number of different TCP/IP interfaces (real or virtual),* the system's IP functions choose which interface to use for the sending IP address, unless told otherwise by a setting of the `interface` option. You may want to use this option if, for example, you are using a lot of IP addresses for web hosting only, and do not want them used for mail. The `interface` option is set to a string that must be an IP address. For example, on a host that has IP addresses 192.168.123.123 and 192.168.9.9, you could set the following:

```
interface = 192.168.123.123
```

in which case all outgoing calls made by the transport would be sent using that particular interface. In a system with IPv6 support, the type of interface specified must be of the same kind as the address to which the call is being made. If not, it is ignored.

* Such hosts are often called *multihomed* hosts.

The `port` option specifies the remote TCP/IP port to which Exim connects in order to send the message. For example:

```
port = 2525
```

If the value begins with a digit, it is taken as a port number; otherwise, it is looked up in */etc/services*. The default setting is `smtp`. This option is mainly used for testing, but is occasionally useful in other circumstances.

By default, Exim sets the socket option `SO_KEEPALIVE` on outgoing socket connections. This causes the kernel to periodically send some *out-of-band* (OOB) data on idle connections. The `no_keepalive` option is provided to disable this, should it ever be necessary. (As far as I know, nobody has ever needed to.)

There are various timeouts associated with SMTP exchanges; normally these work well and you should not need to change them. However, they can be changed by means of the following options (for the last three, the defaults in parentheses are the values recommended in RFC 821):

`connect_timeout`
: Specifies how long to wait for the system's `connect()` function to establish a connection to a remote host. A setting of zero allows the system default timeout (typically several minutes) to act. However, because there have been problems with system default timeouts not working in some operating systems, Exim has a default of 5 minutes. Needless to say, this option has no effect unless its value is less than the system timeout.

`command_timeout` *(5 minutes)*
: Specifies how long to wait for a response to an SMTP command, and also how long to wait for the initial SMTP response after a TCP/IP connection has been established.

`data_timeout` *(5 minutes)*
: Specifies how long to allow for the transmission of one block of message data.* The overall transmission timeout for a message therefore depends on the size of the message.

`final_timeout` *(10 minutes)*
: Specifies how long to wait for a response after the entire message has been sent.

* Exim transmits messages in 8 KB blocks by default.

Use of the SIZE Option in SMTP

If a remote SMTP server indicates that it supports the `SIZE` option of the `MAIL` command, Exim passes over the message size at the start of an SMTP transaction. If the message is too large for the receiving host, it can reject the `MAIL` command, which saves the client from transmitting a large message, only to have it rejected at the end.

The value of the `size_addition` option (default 1024) is added to the size of the message to obtain the argument for `SIZE`. This is to allow for headers and other text that may be added during delivery by configuration options or in a transport filter. It may be necessary to increase this value if a lot of text is added to messages. Alternatively, if the value of `size_addition` is negative, it disables the use of the `SIZE` option altogether.

Use of the AUTH Command in SMTP

When Exim has been built to include support for at least one of the SMTP authentication mechanisms, the `authenticate_hosts` option is available in the **smtp** transport. It is a host list, providing a list of servers to which Exim will attempt to authenticate as a client when it connects, as long as the servers announce authentication support. Details of SMTP authentication are given in Chapter 15, *Authentication, Encryption, and Other SMTP Processing*.

Use of the LMTP Protocol

LMTP (RFC 2033) is a protocol for passing messages between an MTA and a "black box" way of storing mail such as the Cyrus IMAP message store. Later in this chapter, in the section "The lmtp Transport," we discuss the background to LMTP and describe how it can be used to pass messages to local processes. However, LMTP is similar to SMTP, and in some cases there is a requirement to pass messages to a message store over a TCP/IP connection using LMTP instead of SMTP. You can configure Exim to do this by setting up an **smtp** transport with the following option:

```
    protocol = lmtp
```

If you do this, the default value for `port` changes to `lmtp`, but everything else in the transport operates exactly as before. Of course, you must set up a special transport when you do this; LMTP is not used for the normal transmission of messages between MTAs.

Specifying Hosts

There are two options for **smtp** that can specify lists of hosts: `hosts` and `fallback_hosts`.

Specifying a primary host list

The most common situation in which `hosts` is set is when a director (as opposed to a router) is used to cause certain local parts to be delivered remotely. Directors cannot set up host lists, so in this case the list must be defined by the transport. An example of this usage is given in the section "Mixed Local/Remote Domains," in Chapter 5, *Extending the Delivery Configuration*, where we consider a corporate mail gateway that delivers some local parts in its local domain into local mailboxes, and sends others on to personal workstations. The `hosts` setting on the transport specified the workstation.

Overriding a router's host list

When an **smtp** transport is invoked from a router, hosts that are set up by the router normally override hosts set in the transport. That is, the setting of `hosts` in the transport is ignored. Sometimes, however, you may want a router to check for a valid destination, but have the message sent to a different host. If `hosts_override` is set with a host list, it is the router's hosts that are ignored.*

As an example of where this is useful, consider a host permanently connected to the Internet on a slow connection, which sends all outgoing mail to a smart host so that queuing happens on the far side of the slow line. It is useful to be able to check that a remote domain exists before wasting bandwidth sending a message to the smart host. This can be done by using a router such as:

```
lookuphost:
  driver = lookuphost
  transport = smarthost
```

with this transport:

```
smarthost:
  driver = smtp
  hosts = the.smart.host
  hosts_override
```

The router uses the DNS to route addresses in the normal way, so any domains that do not exist fail to be routed and cause their addresses to fail. However, the

* If `hosts_override` is set without a host list, it has no effect.

host list that is set up by the router is ignored by the transport because hosts_override is set, causing all addresses with routeable domains to be delivered to the smart host.

Randomizing a host list

When a hosts setting in an **smtp** transport specifies more than one host, they are tried in the order they are listed, unless hosts_randomize is set. In this case, the order of the list is randomized each time the transport is run. There is no facility for using the hosts in a "round-robin" fashion.

Specifying a fallback host list

The fallback_hosts option provides a "use a smart host only if delivery fails" facility. In the section "Adding Data for Use by Transports," in Chapter 6, we discuss an identically named option for routers and directors. The option on the **smtp** transport has the same effect, but is overridden if fallback hosts are supplied by the router or director. Neither the hosts_override nor the hosts_randomize options apply to fallback_hosts. Once normal deliveries are complete, the fallback queue is delivered by rerunning the same transports with the new host lists. If several failing addresses have the same fallback hosts (and max_rcpt permits it), a single copy of the message is sent to multiple recipients.

Looking up IP addresses

Four options in the **smtp** router control the lookup of IP addresses for hostnames (either from hosts or fallback_hosts). They operate in exactly the same way as the identically named options in the **lookuphost** router (see the section "The lookuphost Router in Chapter 8, *The Routers*, where more detail is given). They are as follows:

gethostbyname
: Requests name lookup by gethostbyname() instead of by calling the DNS resolver.

no_dns_qualify_single
: Turns off the RES_DEFNAMES option of the DNS resolver, which causes it not to qualify domains that consist of just a single component.

dns_search_parents
: Turns on the RES_DNSRCH option of the DNS resolver, causing it to look in parent domains for unknown names.

`mx_domains`
: Provides a list of domain names for which an MX record is required; an address record is not sufficient.

Handling the local host

There is one final option concerned with hosts specified in the **smtp** transport. When a host specified in `hosts` or `fallback_hosts` turns out to be the local host, Exim freezes the message by default. However, if `allow_localhost` is set, it goes on to do the delivery anyway. This should be used only in special cases when the configuration ensures that no looping will result (for example, a differently configured Exim is listening on the port to which the message is sent).

Control of Retrying

A lot of Exim's retrying logic is host-based rather than address- or message-based. That requires it to remember information about failing hosts, and the obvious place to implement the logic for this is in the **smtp** transport, where such failures are detected. There are, as a result, two options concerned with retrying: `retry_include_ip_address` and `delay_after_cutoff`.

Retries are normally based on both the hostname and IP address, so that each IP address of a multihomed host is treated independently. However, in some environments, client hosts are assigned a different IP address each time they connect to the Internet. In this situation, the use of the IP address as part of the retry key on a server host leads to undesirable behavior. Setting `no_retry_include_ip_address` causes Exim to use only the hostname. This should normally be done on a separate instance of the **smtp** transport, set up specially to handle these nonstandard client hosts.

In order to understand `delay_after_cutoff`, you need to know how Exim handles temporary errors and retrying. This is explained in the section "Long-Term Failures," in Chapter 12, *Delivery Errors and Retrying*, where a description of this option can be found.

Summary of smtp Options

The options that are specific to the **smtp** transport are summarized in this section:

`allow_localhost` (Boolean, default = false)
: When a host specified in `hosts` or `fallback_hosts` turns out to be the local host, Exim freezes the message by default. However, if `allow_localhost` is set, it goes on to do the delivery anyway.

`authenticate_hosts` (host list, default = unset)
: This option is available only when Exim is built to contain support for at least one of the SMTP authentication mechanisms. It provides a list of servers to which Exim will attempt to authenticate as a client when it connects, as long as the servers announce authentication support.

`batch_max` (integer, default = 500)
: This controls the maximum number of separate message deliveries that can take place over a single TCP/IP connection. If the value is zero, there is no limit.

`command_timeout` (time, default = `5m`)
: This sets a timeout for receiving a response to an SMTP command that has been sent out. It is also used when waiting for the initial banner line from the remote host. Its value must not be zero.

`connect_timeout` (time, default = `5m`)
: This sets a timeout for the `connect()` function, which sets up a TCP/IP call to a remote host. A setting of zero allows the system timeout (typically several minutes) to operate. To have any effect, the value of this option must be less than the system timeout.

`data_timeout` (time, default = `5m`)
: This sets a timeout for the transmission of each block in the data portion of the message. As a result, the overall timeout for a message depends on the size of the message. Its value must not be zero.

`delay_after_cutoff` (Boolean, default = true)
: This option controls what happens when all remote IP addresses for a given domain have been inaccessible for so long that they have passed their retry cutoff times. See the section "Long-Term Failures," in Chapter 12 for details.

`dns_qualify_single` (Boolean, default = true)
: If the `hosts` or `fallback_hosts` option is being used and names are being looked up in the DNS, the option to cause the resolver to qualify single-component names with the local domain is set if this option is true.

`dns_search_parents` (Boolean, default = false)
: If the `hosts` or `fallback_hosts` option is being used and names are being looked up in the DNS, the resolver option to enable the searching of parent domains is set if this option is true.

`fallback_hosts` (string list, default = unset)
: The value must be a colon-separated list of host names or IP addresses. String expansion is not applied. Fallback hosts can also be specified on routers and directors; these associate such hosts with the addresses they process. Fallback hosts specified on the transport are used only if the address does not have its own associated fallback host list.

`final_timeout` (time, default = `10m`)
This is the timeout that applies while waiting for the response after an entire message has been transported. Its value must not be zero.

`gethostbyname` (Boolean, default = false)
If this option is true when the `hosts` and/or `fallback_hosts` options are being used, names are looked up using `gethostbyname()` instead of using the DNS with MX processing.

`hosts` (string list, default = unset)
This option specifies a list of hosts that are used if the address being processed does not have any hosts associated with it, or if the `hosts_override` option is set.

`hosts_override` (Boolean, default = false)
If this option is set and the `hosts` option is also set, any hosts that are attached to the address are ignored, and instead the hosts specified by the `hosts` option are used.

`hosts_randomize` (Boolean, default = false)
If `hosts_randomize` is false, the order in which hosts are listed is preserved as an order of preference for delivering the message; if it is true, the list specified by `hosts` is shuffled into a random order each time it is used. Randomizing is not performed for fallback hosts.

`interface` (string, default = unset)
This option specifies which local interface to bind to when making an outgoing SMTP call. If `interface` is not set, the system's IP functions choose which interface to use if the host has more than one. In an IPv6 system, the type of interface specified must be of the same kind as the address to which the call is being made. If not, it is ignored.

`keepalive` (Boolean, default = true)
This option controls the setting of `SO_KEEPALIVE` on outgoing socket connections. It causes the kernel to periodically send some out-of-band (OOB) data on idle connections. The option is provided for symmetry with the main configuration option `smtp_accept_keepalive`, which has the same effect on incoming SMTP connections.

`max_rcpt` (integer, default = 100)
This option limits the number of `RCPT` commands that are sent in a single SMTP message transaction. Each set of addresses is treated independently, and so can cause parallel connections to the same host if `remote_max_parallel` permits this.

`multi_domain` (Boolean, default = true)
When this option is set, the **smtp** transport can handle a number of addresses containing a mixture of different domains provided they all resolve to the same list of hosts. Turning the option off restricts the transport to handling only one domain at a time.

`mx_domains` (domain list, default = unset)
If the `hosts` or `fallback_hosts` options are being used and names are being looked up in the DNS, any domain name that matches this list is required to have an MX record; an address record is not sufficient.

`port` (string, default = `smtp`)
This option specifies the TCP/IP port that is used to send the message. If it begins with a digit, it is taken as a port number; otherwise, it is looked up using `getservbyname()`.

`protocol` (string, default = `smtp`)
If this option is set to `lmtp` instead of `smtp`, the default value for the `port` option changes to `lmtp`, and the transport uses the LMTP protocol instead of SMTP.

`retry_include_ip_address` (Boolean, default = true)
Setting this option false causes Exim to use only the hostname instead of both the name and the IP address when constructing retry records. Details of how this works are given in the section "Retrying After Errors," in Chapter 12.

`serialize_hosts` (host list, default = unset)
This option lists the hosts to which only one TCP/IP connection at a time should be made.

`size_addition` (integer, default = 1024)
The value of `size_addition` is added to the value Exim sends in the SMTP `SIZE` option to allow for headers and other text that may be added during delivery by configuration options or in a transport filter. It may be necessary to increase this if a lot of text is added to messages. If the value of `size_addition` is negative, the use of the `SIZE` option is disabled.

Environment for Local Transports

Local transports handle deliveries to files and pipes. (The **autoreply** transport can be thought of as similar to a pipe.) Whenever a local transport is run, Exim forks a subprocess for it. Before running the transport code, it sets a specific uid and gid.*

* This assumes a conventional Exim installation, where Exim is privileged by virtue of being a setuid binary. See the section "Running an Unprivileged Exim," in Chapter 19, *Miscellany*, for a discussion of unconventional configurations where this is not true.

This ensures that local deliveries are done "as the user," so that access to files and programs is controlled by the normal operating system protection mechanism.

Exim also sets a current file directory; for some transports, a home directory setting is also relevant. The **pipe** transport is the only one that sets up environment variables; see the section "The pipe Transport," later in this chapter for details.

The values used for the uid, gid, and the directories may come from several different places. In many cases, the director that handles the address associates settings with that address. However, values may also be given in the transport's own configuration, and these override anything that comes with the address. The sections below contain a summary of the possible sources of the values and how they interact with each other.

Uids and Gids

All local transports have the options `group` and `user`. If `group` is set, it overrides any group that may be set in the address, even if `user` is not set. This makes it possible, for example, to run local mail delivery under the uid of the recipient, but in a special group, use a transport such as this:

```
group_delivery:
  driver = appendfile
  file = /var/spool/mail/$local_part
  group = mail
```

You might want to do this if all the mailbox files are precreated and set up so that group *mail* can write to them. See the section "Mailbox location," later in this chapter for further discussion of this.

This example assumes that the director has set a value for the user (which will be the case if it is the standard **localuser** director), but it overrides any group setting that the director may have made. Likewise, if `user` is set for a transport, its value overrides what is already set in the address. If `user` is nonnumeric and `group` is not set, the gid associated with the user is used. If `user` is numeric, `group` must be set.

Every Unix process runs under a specific uid and gid, but in addition, a number of other groups can be associated with it. These are held in the *supplementary group access list*, and give the process privileges that are associated with those groups. For example, users who are members of the groups *staff*, *network*, and *admin* have all these groups set up in their login processes. Setting up the supplementary groups list uses resources, and is typically not needed for email delivery, so Exim does not do it by default.

For delivery via a pipe, however, you may sometimes want Exim to set up the supplementary groups list. The **pipe** transport has an `initgroups` option that lets you request this, but only when `user` is also specified on the transport. When the user is associated with the address by a director or router, the value of the `initgroups` option is taken from the director or router configuration.

Current and Home Directories

The **pipe** transport has a `home_directory` option. If this is set, it overrides any home directory set by the director for the address. The value of the home directory is set in the environment variable `HOME` while running the pipe. It need not be set, in which case `HOME` is not defined.

The **appendfile** transport does not have a `home_directory` option. If the variable $home appears in one of its options, the value set by the director is used. This also applies to the `inhome` or `belowhome` settings of the `create_file` option.

The **appendfile** and **pipe** transports have a `current_directory` option. If this is set, it overrides any current directory set by the director for the address. If neither the director nor the transport sets a current directory, Exim uses the value of the home directory, if set. Otherwise it sets the current directory to the root directory before running a local transport. All directors have `current_directory` and `home_directory` options, which are associated with any addresses they explicitly direct to a local transport.

Routers have no means of setting up home and current directory strings; consequently, any local transport that they use must specify them for itself if they are required.

Expansion Variables Derived from the Address

Normally, a local delivery handles a single address, and it sets variables such as $domain and $local_part during the delivery. However, local transports can be configured to handle more than one address at once (for example, while writing a batch SMTP for onward transmission by some other means). In this case, the variables associated with the local part are never set, $domain is set only if all the addresses have the same domain, and $original_domain is never set.

Options Common to the appendfile and pipe Transports

The **appendfile** transport, which writes messages to files, and the **pipe** transport, which writes messages to pipes that are connected to other processes, have a number of options in common. To save repetition, they are collected together in this section.

Controlling the Delivery Environment

The three options current_directory, group, and user can be used to control the environment in which the local delivery process for these transports runs. Details of these options are given in the section “Environment for Local Transports,” earlier in this chapter.

Controlling the Format of the Message

As the message is written to a file or down a pipe, certain modifications can be made by the transport. Generic options for adding or removing header lines are covered in the section “Options Common to All Transports,” earlier in this chapter. Here we describe some options that apply only to local deliveries.

Separating messages in a single file

When a mailbox consists of a single file that contains concatenated messages, there has to be some way of determining where one message ends and another begins. Several schemes have been used for this, the most common of which is the *Berkeley mailbox format*, in which a message begins with a line starting with the word From, followed by a space and other data, and ends with an empty line.

This format seems to have been adopted in the distant past because messages received by UUCP use such a line to contain the envelope sender address.* There is, however, no standard for it as a message separator, and several variants have been seen in practice. It is a most unfortunate choice of separator line, because lines within the bodies of messages that start with the word From are not at all uncommon. This means that such lines have to be escaped in some way (as described shortly), lest they be taken as the start of a new message.

Exim supports message separation by providing two options called prefix and suffix. Their contents are expanded and written at the start and end of every message, respectively. The default values for the **appendfile** transport are:

```
prefix = "From ${if def:return_path{$return_path}{MAILER-DAEMON}} \
         $tod_bsdinbox\n"
suffix = "\n"
```

These define Berkeley format message separation. The prefix setting places the envelope sender (return path) in the separator line, unless the message is a bounce message (where there is no return path). For bounce messages, MAILER-DAEMON is used, because some programs that read mailboxes do not work if nothing is inserted. The line ends with the date and time in a particular format required

* See RFC 976, *UUCP Mail Interchange Format Standard.*

by this form of separator, which is made available in the variable $tod_bsdinbox. The `suffix` setting ensures that there is a blank line after every message.*

The same defaults are used in the **pipe** transport (though message separation is not an issue in this case), because some implementations of the commonly used */usr/ucb/vacation* command expect to see a `From` line at the start of messages that are piped to them. However, in many applications of the **pipe** transport, the `From` line is not expected and should be disabled. An example of this is given in the section "Using an External Local Delivery Agent," in Chapter 5.

The Berkeley format is not the only one that uses textual separators between messages. In MMDF format, the beginnings and ends of messages are marked by lines containing exactly four nonprinting characters whose numeric code value is 1. The **appendfile** transport can be configured to support this by setting the following:

```
prefix = "\1\1\1\1\n"
suffix = "\1\1\1\1\n"
```

If you want to use such a format, you must be sure all your MUAs can interpret it.

Escaping lines in the message

The mechanism provided for handling lines in the message that happen to look like message separators is a pair of options called `check_string` and `escape_string`. The default settings in the **appendfile** transport are:

```
check_string = "From "
escape_string = ">From "
```

But for the **pipe** transport, both of these options are unset by default. As the transport writes the message, the start of each line is tested for matching `check_string`; if it does, the initial matching characters are replaced by the contents of `escape_string`. The value of `check_string` is a literal string, not a regular expression. The default therefore inserts a single angle-bracket character before any line starting with `From` followed by a space. For example, if a message contains the line:

```
From the furthest reaches of the Galaxy, ...
```

appendfile actually writes:

```
>From the furthest reaches of the Galaxy, ...
```

* The message itself ends with a newline, because the SMTP protocol is defined in terms of lines. For messages submitted locally, Exim adds a final newline if there isn't one.

If you are using MMDF mailbox separators, change the default settings to:*

```
check_string  = "\1\1\1\1\n"
escape_string = "\1\1\1\1 \n"
```

When batch SMTP delivery is configured (see the section "A BSMTP batching example," later in this chapter), the contents of `check_string` and `escape_string` are forced to values that implement the SMTP escaping protocol for lines beginning with a dot. Any settings made in the configuration file are ignored.

Control of line terminators

There is one more option that affects the contents of a message. It is `use_crlf`, and when it is set, lines are terminated by CRLF instead of just a linefeed. Some external local delivery programs require this. It may also be useful in the case of batched SMTP, because the byte sequence written to the file is then an exact image of what would be sent down a real SMTP connection, as long as you remember to unset `prefix` and `suffix`. Note that if you do use this option with `prefix` or `suffix`, you need to ensure that any occurrence of `\n` in those strings is changed to `\r\n` because they are written verbatim, without any interpretation.

Batched Delivery and BSMTP

When **appendfile** and **pipe** are used in their traditional way, each delivery is for a single recipient only. If, for example, a message is to be delivered into several local mailboxes, **appendfile** is run once for each mailbox, and the same is true for deliveries to pipes. However, there are special circumstances in which it is helpful to transport a single copy of a message for several different recipients. The most common case is for messages that have not reached their final destination, but are going to be transported further by some means outside Exim. For example:

- Messages for dial-up hosts are commonly stored in files, often in a per-host directory, and transmitted to the hosts when they connect. See the section "Incoming Mail for an Intermittently Connected Host," in Chapter 12 for more discussion of this topic.
- Messages for other transport mechanisms such as UUCP can be passed using pipes to programs that support such mechanisms.

If the `batch` option is set to a string other than `none` (the default), the addresses that are routed or directed to the transport can be combined into deliveries with multiple addresses. However, there are some other conditions that must be met for this to happen.

* If a message contains binary data, changing it in this way may damage the data, but that is probably better than a broken mailbox file.

- If the configuration of the transport refers to $local_part,then no batching is possible.
- If `batch` is set to `domain`, or if the configuration of the transport refers to $domain, only addresses that have the same domain can be batched.
- If `batch` is set to `all` and the configuration does not refer to $domain, all addresses can be batched, even if they have different domains.
- The maximum number of addresses in one batch is set by `batch_max`, which defaults to 100.
- The batched addresses must all have the same return paths, requirements for header removal and addition, and $route_option and $host settings (for routed addresses), and they must specify the same uid and gid for delivery. In other words, batching several addresses into a single delivery can take place only when the copy of the message is identical and the delivery environment is the same for all those addresses.

The $local_partvariable is never set when more than one address is being transported, and $domainis set only if all addresses have the same domain. If the generic `envelope_to_add` option is set, all the addresses are included in the *Envelope-to:* header line that is added to the message.

A non-BSMTP batching example

Suppose you wanted to collect all messages for certain remote domains in files, instead of sending them out over TCP/IP. First, set up a router that picks off the domains and routes them to a special transport:

```
filed_domains:
  driver = domainlist
  transport = filed_domains
  route_list = first.filed.example  first ;\
               second.filed.example second
```

Now set up the transport, which in this example uses the hostname taken from the second field in the routing rule to generate the filename:

```
filed_domains:
  driver = appendfile
  file = /var/savedmail/$host
  envelope_to_add
  return_path_add
  user = mail
```

There has to be a setting of `user` either on the router or the transport, to specify the uid under which the delivery process runs. Setting `envelope_to_add` and `return_path_add` ensures that the relevant parts of the message's envelope are preserved with the message.

A BSMTP batching example

An alternative way of preserving message envelopes is to use `bsmtp` instead of `batch`. This causes messages to be written as if they were being transmitted over an SMTP connection, and is normally referred to as *batch SMTP* or BSMTP. Each message starts with a `MAIL` command for the envelope sender, followed by a `RCPT` command for each recipient, and then `DATA` and the message itself, terminated by a dot. For example, a file written this way could contain:

```
MAIL FROM:<tom@abcd.example>
RCPT TO:<jerry@pqrs.example>
RCPT TO:<bugs@albuquerque.example>
DATA
<message content>
.
```

The `bsmtp` option can be set to the same strings as `batch`, with one addition: if it is set to `one`, it has the same effect as setting `batch_max` to 1 (that is, a separate delivery is done for each recipient).

Some programs that read BSMTP files expect that each message is preceded by a `HELO` command. Exim inserts such a command if `bsmtp_helo` is set, but not otherwise.

The settings of the `prefix` or `suffix` options are not affected by `batch` or `bsmtp`, and if they are set, their contents are included in the output. Normally this is not wanted for BSMTP, so a typical transport might look like this:

```
filed_domains:
  driver = appendfile
  file = /var/savedmail/$host
  bsmtp = all
  bsmtp_helo
  prefix =
  suffix =
  user = mail
```

On the other hand, the settings of `check_string` and `escape_string` are affected by `bsmtp` (but not `batch`). They are forced to the following values:

```
check_string = .
escape_string = ..
```

in order to implement the standard SMTP-escaping mechanism.

Use of multiple files for batched messages

The **appendfile** examples given in this section assume that multiple messages to the same address are being written to a single file. This does not have to be the case; **appendfile** can operate by writing each message as a separate file (see the section "Delivering Each Message into a Separate File," later in this chapter). The batching options are independent of this choice.

Control of Retrying

When a local delivery suffers a temporary failure, both the local part and the domain are normally used to form a key that is used to determine when next to try the address. This handles common cases such as exceeding a mailbox quota, where the failure applies to the specific local part. However, when local delivery is being used to collect messages for onward transmission by some other means, a temporary failure may not depend on the local part at all, so a temporary failure for *any* local part should cause other deliveries for the same domain to be delayed according to the retry rules. Setting `no_retry_use_local_part` causes Exim to use only the domain when handling retries for the transport on which it is set.

Summary of Options Common to appendfile and pipe

The options that are common to the **smtp** and **pipe** transports are summarized in this section:

`batch` (string, default = `none`)
: If this option is set to the string `domain`, all addresses with the same domain that are directed or routed to the transport are handled in a single delivery. If it is set to `all`, then multiple domains can be batched.

`batch_max` (integer, default = 100)
: This limits the number of addresses that can be handled in a batch and applies to both the `batch` and the `bsmtp` options.

`bsmtp` (string, default = `none`)
: This option is used to set up an **appendfile** or **pipe** transport for delivering messages in batch SMTP format. It is usually necessary to suppress the default settings of the `prefix` and `suffix` options when using batch SMTP. The value of `bsmtp` must be one of the strings `none`, `one`, `domain`, or `all`. The first of these turns the feature off.

`bsmtp_helo` (Boolean, default = false)
: When this option is set, a `HELO` line is added to the output at the start of each message written in batch SMTP format.

`check_string` (string, default = see description)
: As the transport writes the message, the start of each line is tested for matching `check_string`, and if it does, the initial matching characters are replaced by the contents of `escape_string`. The value of `check_string` is a literal string, not

a regular expression. For the **appendfile** transport, the default is `From` followed by a space, whereas the default is unset for the **pipe** transport.

`current_directory` (string, default = unset)
: If this option is set, it specifies the directory to make current when running the delivery process. The string is expanded at the time the transport is run.

`escape_string` (string, default = `>From`)
: See `check_string`.

`prefix` (string, default = see description)
: The string specified here is expanded and output at the start of every message. The default setting is:

```
    prefix = "From ${if def:return_path{$return_path}\
      {MAILER-DAEMON}} ${tod_bsdinbox}\n"
```

`retry_use_local_part` (Boolean, default = true)
: Setting this option to false causes Exim to use only the domain when handling retries for this transport.

`suffix` (string, default = `\n`)
: The string specified here is expanded and output at the end of every message.

`use_crlf` (Boolean, default = false)
: This option causes lines to be terminated with the two-character CRLF sequence (carriage return, linefeed) instead of just a linefeed character.

The appendfile Transport

Writing messages to files is the most complex operation of Exim's local transports; consequently, **appendfile** has a large number of options. This transport can operate in two entirely different modes:

- In "file" mode, the message is appended to the end of a file, which may pre-exist and contain other messages. Two major message formats are supported, with minor variations controllable by the `prefix` and `suffix` options.
- In "directory" mode, the message is written to an entirely new file within a specific directory. Three different file formats are supported.

When writing to a single file containing multiple messages, the file has to be locked so that neither MUAs nor other Exim processes can tamper with it while the delivery is taking place. This means that only one message can be delivered to this mailbox at any one time, and while it is being delivered, messages that are already in the mailbox cannot be removed. If, on the other hand, each message is written to a new file, no locking is required, multiple simultaneous deliveries can take place, and old messages can be deleted at any time.

Setting Up a Multimessage File for Appending

When **appendfile** is called as a result of a filename item in a user's *.forward* file or as the result of a *save* command in a filter file, the director passes the name of the file to the transport, along with the address that is being delivered. In other cases, when no filename is already associated with the address, the `file` option specifies the name of the file to which the message is to be appended. An example that has been used several times for delivery to conventional mailboxes is:

```
file = /var/mail/$local_part
```

where the name of the file depends on the local part that is being delivered. This is a very simple example, but if necessary, the full power of string expansions can be used to compute the filename.

Mailbox location

On hosts where the users have login access, some installations choose to locate users' mailboxes in their home directories. This has three benefits:

- There are no file permission complications, because users have full access to their own home directories.
- If there are a large number of users, you avoid the problem of a lot of files in one directory.
- The size of users' mailboxes can be constrained by the system file quota mechanism.

If you want to configure Exim this way, you can use a setting of the `file` option like this:

```
file = /home/$local_part/inbox
```

The disadvantages of this approach are as follows:

- You may have to modify, or at least recompile user agents so that they know where to find users' mailboxes.
- If the home directories are automounted, there will be additional mounts and dismounts, because the home directory has to be mounted each time a new message arrives. On the other hand, if you allow users to have *.forward* files in their home directories, these mounts will occur anyway.

The alternative, and probably more common approach, is to locate all the mailboxes in a single directory, separate from the home directories. The directory is usually called */var/mail* or */var/spool/mail*, and most user agents expect to find mailboxes in one of these directories by default. This scheme works well enough for a moderate number of mailboxes, but when the number gets large you normally have to find some way of splitting up the directory to avoid performance loss. See the section "Large Installations," in Chapter 4, *Exim Operations Overview*, for a discussion of this point.

With all the mailboxes in a single directory, there has to be a way for an Exim delivery process, running as the recipient user, to create a new mailbox if it does not exist, and to be able to write to an existing mailbox.* Allowing all users full access to the directory is not the answer, because that would let one user delete another user's mailbox. Unix contains a file permission facility that is intended for just this situation. For historical reasons, it is known as the *sticky bit.* When a directory has this permission set, along with the normal write permission, any user may create a new file in the directory, but files can be deleted only by their owners. The letter `t` is used to indicate this permission, so a mailbox directory that is set up this way looks like the following:

```
drwxrwxrwt   3 root     mail         512 Jul  9 13:48 /var/mail/
```

Exim's local delivery processes, which run using the uid and gid of the receiving user, can now create new mailboxes if necessary. Each mailbox will be owned by the local user, and by default, will be accessible only to that user, who can modify it and also delete it.

There are some disadvantages to using "sticky" directories:

- Users may try to use the directory as extra space to store files that are not mailboxes.
- When a user's mailbox does not exist, a different user may maliciously create it, hoping to be able to access the first user's mail. This does not work, because Exim checks the ownership of existing files, but it does prevent the first user from receiving mail.

Installations that find these possibilities unacceptable often adopt a different approach. Instead of using a "sticky" directory, they make use of the group access instead. The directory's permissions are set in the following manner:

```
drwxrwx---   3 root     mail         512 Jul  9 13:48 /var/mail/
```

The **appendfile** transport is configured to run under the group *mail* instead of the user's group, like this:

```
local_delivery:
  driver = appendfile
  file = /var/mail/$local_part
  group = mail
  mode = 660
```

* Exim may also need to create and remove lock files. See the section "Locking a File for Appending," later in this chapter.

It still runs with the user's uid, however. With this configuration, Exim can create new mailboxes because of the group write permission. Each mailbox is owned by the relevant user, but its group is set to *mail*. Exim can update an existing mailbox because the `mode` setting allows group *mail* to write to the file. The user's MUA can access the file as the owner. Nevertheless, there is still a disadvantage: the user cannot delete the file. Some MUAs do try to delete the file when all the messages it contains have been deleted or moved, which may give rise to error messages.

Symbolic links for mailbox files

By default, **appendfile** will not deliver a message if the pathname for the file is that of a symbolic link. Setting `allow_symlink` relaxes that constraint, but there are security issues involved in the use of symbolic links. Be sure you know what you are doing if you allow them. The ownership of the link is checked, and the real file is subjected to the checks described in the section "The owner of an existing file," and in the section "The mode of the file." The check on the top-level link ownership prevents one user from creating a link for another's mailbox in a "sticky" directory, though allowing symbolic links in this case is definitely not a good idea. If there is a chain of symbolic links, the intermediate ones are not checked.

Delivering to named pipes (FIFOs)

As with symbolic links, **appendfile** will not deliver to a FIFO (named pipe) by default, but can be configured to do so by setting `allow_fifo`. If no process is reading the named pipe at delivery time, the delivery is deferred.

Creating a nonexistent file

The default action is to try to create the file and any superior directories if they do not exist. Several options give control over this process:

- If `file_must_exist` is set, an error occurs if the file does not exist, the delivery is deferred, and the message is frozen.
- Otherwise, the value of `create_file` is inspected for constraints on where the file may be created. The option can be set to one of the following values:
  - `anywhere` means there is no constraint.
  - `inhome` means that the file may be created only if it is in the home directory.
  - `belowhome` means that the file may be created in the home directory or any directory below the home directory.

  In the second and third cases, a home directory must have been set up for the address by the director that handled it; this is the normal case when user forwarding or filtering is involved. The `create_file` option is not useful when an explicit filename is given for normal mailbox deliveries; it is intended for the case when filenames have been generated by user forwarding or filtering. In addition to this constraint, the file permissions must also permit the file to be created, of course. Remember that a local delivery always runs under some unprivileged uid and gid.

- The `create_directory` option controls whether superior directories may be created. It is true by default, but creation of directories occurs only when creation of the file is also permitted.

The owner of an existing file

If a file already exists, checks are made on its ownership and permissions. The owner must be the uid under which the delivery process is running, unless `no_check_owner` is set. The uid is either set by the `user` option on the transport, or passed over with the address by the director or router. For the common cases of delivery into user mailboxes and deliveries to files specified in *.forward* files, the user is normally the local user that corresponds to the local part of the address. If `check_group` is set, the group ownership of the file is also checked. This is not the default, because the default file mode is 0600 (owner read/write only), for which the group is not relevant.

The mode of the file

If the delivery is the result of a *save* command in a filter file that specifies a particular mode for the file, a new file is created with that mode, and an existing file is changed to have that mode. Otherwise, if the file is created, its mode is set to the value specified by the `mode` option, which defaults to 0600. If the file already exists and has wider permissions (more bits set) than those specified by `mode`, they are reduced to the value of `mode`. If it has narrower permissions, an error occurs and the message is frozen unless `no_mode_fail_narrower` is specified, in which case the delivery is attempted with the existing mode.

Format of Appended Messages

The default way of appending a message to a file is to write the contents of the `prefix` option, followed by the message, followed by the contents of `suffix` (see the section "Controlling the Format of the Message," earlier in this chapter). These default values support traditional Berkeley Unix mailboxes:

```
prefix = "From ${if def:return_path{$return_path}{MAILER-DAEMON}} \
         $tod_bsdinbox\n"
suffix = "\n"
```

Other formats that rely on textual separators can be used by changing `prefix` and `suffix`, as was shown in the MMDF example in the section "Separating messages in a single file," earlier in this chapter.

MBX format mailboxes

Another format that is supported by **appendfile** is MBX, which is requested by setting `mbx_format`.* This single-file mailbox format is supported by Pine 4 and its associated IMAP and POP daemons, and is implemented by the c-client library that they all use.

There is additional information about the lengths of messages at the beginning of an MBX mailbox. This makes it faster to access individual messages. Simultaneous shared access to MBX mailboxes by different users is also possible. However, a special form of file locking is required, and `mbx_format` should not be used if any program that does not use this form of locking is going to access the mailbox. MBX locking cannot be used if the mailbox file is NFS-mounted, because this type of locking works only when the mailbox is accessed from a single host. See the section "Locking options for MBX mailboxes," later in this chapter, for more discussion of MBX locking.

In order to maintain a mailbox in MBX format, Exim has to write the message to a temporary file before appending it so that it can obtain its exact length, including any header lines that are added during the delivery process. This makes this form of delivery slightly more expensive.

The `prefix`, `suffix`, and `check_string` options are not automatically changed by the use of `mbx_format`; they should normally be set empty because MBX format does not rely on the use of message separators. Thus, a typical **appendfile** transport for MBX delivery might look like this:

```
local_delivery:
  driver = appendfile
  file = /home/$local_part/inbox
  delivery_date_add
  envelope_to_add
  return_path_add
  mbx_format
  prefix =
  suffix =
  check_string =
```

* The code for this is not built into Exim by default, and has to be requested in the build-time configuration.

Checking an existing file's format

When Exim is adding a message to an existing mailbox file, the file is assumed to be in the correct format because normally only a single format is used on any one host. Sometimes, however, particularly during a transition from one format to another, files in different formats may coexist. In these situations, **appendfile** can be made to check the format of a file before writing to it, and if necessary, it can pass control to a different transport. For example, suppose that both Berkeley and MBX mailboxes exist on the system. The following could be added to the `local_delivery` transport defined earlier:

```
file_format = "*mbx*\r\n : local_delivery :\
               From      : local_bsd_delivery"
```

The items in a `file_format` list are taken in pairs. The first of each pair is a text string that is compared against the characters at the start of the file. If they match, the second string in the pair is the name of the transport that is to be used. In this example, if the file begins with `*mbx*\r\n`, the **local_delivery** transport is required. As this is the current transport, delivery proceeds. If, however, the file begins with `From`, control is passed to a transport called **local_bsd_delivery**, which might be defined thus:

```
local_bsd_delivery:
  driver = appendfile
  file = /home/$local_part/inbox
  delivery_date_add
  envelope_to_add
  return_path_add
```

If the file does not exist or is empty, the format used by the first mentioned transport is use, so in this example new files are created in MBX format. If the start of a file does not match any string, or if the transport named for a given string is not defined, delivery is deferred.

Locking a File for Appending

In most Exim configurations, the parameters for controlling file locking can be left at their default values. Unless you want to understand the nitty-gritty of locking or have a special locking requirement, you can skip this section.

Mailbox files are modified by both MTAs and MUAs. MTAs add messages to files, but MUAs can both add and remove messages, though deletion is their most common action. When an MTA is updating a file by appending a message, it must ensure that it has exclusive control of the file, so that no other process can attempt to update it at the same time. Because of Exim's distributed nature, it is possible for several deliveries to the same file to be running simultaneously so Exim has to coordinate its own processes, as well as lock out any MUAs that may be operating on the file.

Locking in Unix is a purely voluntary action on the part of a process. Its success relies on cooperative behavior among all the processes that access a shared file. Two different conventions have arisen for locking mailbox files, and because it cannot assume that the MUAs on a system all use the same one, Exim normally obtains both kinds of lock before updating a file.

Locking using a lock file

The first method of locking is to use a *lock file*. Unix contains a primitive operation that creates a file if it doesn't exist, or returns an error if it does exist. A process that needs exclusive access to a file called */a/b/c*, for example, attempts to create a file called */a/b/c.lock* by this method.* If it succeeds, it has ownership of the lock; if it fails because the file already exists, some other process has the lock, and the first process must wait for a while and then try again. Once a process has finished updating the original file, it deletes the lock file. For this scheme to work, the following conditions must be met:

- All processes must use the same name for the lock file.
- The users running the processes must be able to create a new file (the lock file) in the directory that contains the original file.

There are two main problems with lock files:

- If a process crashes while it holds a lock, the lock file is not deleted. For this reason, most implementations use some kind of timeout and force the deletion of lock files that are too old.
- A process that is waiting for a lock has to wait for a fixed time before retrying; it cannot put itself onto some kind of queue and be automatically woken up when the lock becomes available.

Despite these problems, lock files are important because they are the only reliable way of locking that works over NFS when more than one host is accessing the same NFS file system. This is because the use of a lock file enables a process to obtain a lock before opening the shared file itself.

Using a locking function

The other method of locking operates only on an open file. For historical reasons, there are several different Unix calls for doing this; Exim uses the `fcntl()`

* In practice, to cater for use over NFS, it is not quite as simple as just attempting to create the file, but the principle is the same.

function.* This method of locking does not suffer from the problems of lock files because:

- No second filename is needed.
- The ability to create a new file is not needed.
- If a process crashes, the files it has open are automatically closed and their locks released.
- A call to lock a file can be queued, so that the lock is obtained as soon as it becomes available.

Why use lock files?

Why isn't everybody using `fcntl()` locks exclusively, since they seem to be much better than lock files? One reason is history: lock files came first, and some older programs may still not be using anything else. A much more important reason is the widespread use of NFS. When more than one host is accessing the same NFS file system, `fcntl()` locks do not work. The reason is that the size of a file is saved in the NFS client host when the file is opened, so if two clients open a file simultaneously, they both receive the same size; however, if one then obtains an `fcntl()` lock and updates the file while the other waits, the second one will have an incorrect size when it eventually gets the lock. To update NFS files safely from different hosts, it is necessary to obtain a lock before opening the file, and lock files provide the only way of doing this.

Locking options for non-MBX mailboxes

The **appendfile** transport supports both kinds of lock for traditional mailbox formats, as well as a third variety for MBX mailboxes, which we will cover shortly. The default settings are to use both lock files and `fcntl()` locking to cater for all possible situations, but options are provided for selecting one or the other kind of lock and for changing some of the locking parameters. The mode of any created lock file is set by `lockfile_mode`.

If `use_lockfile` is set false, lock files are not used, and if `use_fcntl_lock` is set false, locking by `fcntl()` is disabled. Only one kind of locking can be turned off at once, but in most cases you should not do either of these things. You should only contemplate turning one of them off if you are absolutely sure that the remaining locking is sufficient for all programs that access the mailbox.

When lockfiles are in use, failure to create a lockfile through lack of permission is not treated as an error if `require_lockfile` is set false. When this is the case, the

* This interworks with the `lockf()` function, but not with the older `flock()`.

delivery relies on `fcntl()` locking, so again you should disable this option only if you are sure that it will not cause problems.

Various timeouts are used when attempting to lock a mailbox, and their values can be changed if necessary. If an existing lock file is older than `lockfile_timeout`, Exim assumes it has been left behind by accident, and attempts to remove it.

By default, nonblocking calls to `fcntl()` are used. If a call fails, Exim sleeps for `lock_interval`, and then tries again, up to `lock_retries` times. Nonblocking calls are used so that the file is not kept open during the wait for the lock; the reason for this is to make it as safe as possible for NFS deliveries in the case when processes might be accessing an NFS mailbox without using a lock file.* On a busy system, however, performance is not as good as using a blocking lock with a timeout. If `lock_fcntl_timeout` is set to a nonzero time, blocking locks with that timeout are used. There may still be some retrying: the maximum number of retries is computed as:

```
    (lock_retries * lock_interval) / lock_fcntl_timeout
```

rounded up to the next whole number. In other words, the total time during which **appendfile** is trying to get a lock is roughly the same with blocking or nonblocking locks, unless `lock_fcntl_timeout` is set very high.

Locking options for MBX mailboxes

When **appendfile** is configured for MBX mailboxes (by setting `mbx_format`), the default locking rules are different. If none of the locking options are mentioned in the configuration, `use_mbx_lock` is assumed, and the other locking options default to false. It is possible to use the other kinds of locking with `mbx_format`, but `use_fcntl_lock` and `use_mbx_lock` are mutually exclusive. MBX locking follows the locking rules of the c-client library, which are explained here.

Exim takes out a shared `fcntl()` lock on the mailbox file, and an exclusive lock on the file whose name is */tmp/.device-number.inode-number*, where the device and inode numbers are those of the mailbox file. The shared lock on the mailbox stops any other MBX client from getting an exclusive lock on it and expunging it (removing messages). The exclusive lock on the */tmp* file prevents any other MBX client from updating the mailbox in any way. When writing is finished, if an exclusive lock on the mailbox itself can be obtained, indicating there are no current sharers, the */tmp* file is unlinked (deleted). Otherwise it is left, because one of the other sharers might be waiting to lock it. The `fcntl()` calls for getting these locks are nonblocking by default, but can be made to block by setting `lock_fcntl_timeout`.

* This should not be done, but accidents do happen, and an inexperienced administrator might end up with such a configuration.

MBX locking interoperates correctly with the c-client library, providing for shared access to the mailbox. It should not be used if any program that does not use this form of locking is going to access the mailbox, nor should it be used if the mailbox file is NFS-mounted, because it works only when the mailbox is accessed from a single host.

If you set `use_fcntl_lock` with an MBX-format mailbox, you cannot use an MUA that uses the standard version of the c-client library, because as long as it has a mailbox open (this means for the whole of a Pine or IMAP session, for example), Exim is unable to append messages to it.

Delivering Each Message into a Separate File

So far, we have been considering cases where a mailbox consists of a single file and new messages are delivered by appending. This is by far the most common type of configuration, but there is an alternative, which is to deliver each message into a separate new file. The mailbox then consists of an entire directory, with each message in its own file. There are advantages and disadvantages to this approach. The advantages are as follows:

- No locking of any kind is required when delivering. A consequence of this is that more than one message can be delivered into the mailbox simultaneously, which is beneficial on very busy systems.
- Deleting messages is easy and quick, since it does not require any rewriting, and it is not necessary to hold up new deliveries while it happens.
- Incomplete deliveries (which can arise as a result of a system failure, for example) can be handled better.
- The problems concerning message separators that arise in single-file mailboxes are all bypassed.

The disadvantages are as follows:

- If a mailbox contains a large number of messages, the number of files in the directory may impact performance.*
- Most common MUAs support only the single-file type of mailbox.

As a consequence of the last point, systems that use this kind of delivery for local mailboxes are usually ones where access to the mailboxes is carefully controlled, for example, by providing only POP or IMAP access.† However, delivery into separate files is also commonly used in an entirely separate situation: namely, when

* Different operating systems, and indeed different filesystems, have different thresholds beyond which performance degrades.

† The POP and IMAP daemons that are used must, of course, be able to handle multifile mailboxes.

messages are being stored temporarily for a dial-up host. This topic is pursued further in the section "Intermittently Connected Hosts," in Chapter 12.

The **appendfile** transport can be configured to deliver each message into a separate file by replacing the `file` option with the `directory` option. The setting of `create_directory` determines whether the directory may be created if it does not exist, and if it is created, `directory_mode` specifies its mode. The data that is written is identical to the single-file mailbox case; in particular, `prefix` and `suffix` data is written if configured, and `check_string` is respected. Normally these features are not required when each message is in its own file, so they should usually be turned off.

Three different separate-file formats are available, but only the most commonly used one is discussed here, because the others are not widely used.

Maildir Format

A separate-file delivery mode for local mailboxes that is gaining general acceptance (not just with Exim) is called *maildir*. This is more complicated than just delivering into individual files, and it operates as follows:

- Within the designated directory, three subdirectories called *tmp*, *new*, and *cur* are created.
- A message is delivered by writing it to a new file in *tmp*, and then renaming it into the *new* directory. The filename usually includes the hostname, the time, and the process ID, but the only requirement is that it be unique.
- Once a mail-reading program has seen a new message in *new*, it normally moves it to *cur*. This reduces the number of files it has to inspect when it starts up.

Files in the *tmp* directory that are older than, say, 36 hours should be deleted; they represent delivery attempts that failed to complete.

Exim delivers in maildir format if `maildir_format` is set. This is a typical configuration:

```
maildir_delivery:
  driver = appendfile
  directory = /home/$local_part/maildir
  delivery_date_add
  envelope_to_add
  return_path_add
  maildir_format
  prefix =
  suffix =
  check_string =
```

The presence of `directory` rather than `file` specifies one file per message, and `maildir_format` specifies that the maildir rules should be followed. Note the explicit cancelling of `prefix`, `suffix`, and `check_string`. Messages delivered by this transport end up in files with names such as:

```
/home/caesar/maildir/new/955429480.9324.host.example
```

An additional string can be added to the name by setting `maildir_tag`. The contents of this option are expanded and added to the filename when the file is moved into the *new* directory. The tag may contain any printing characters except a forward slash; if it starts with an alphanumeric character, a leading colon is inserted. By this time, the file has been written to the *tmp* directory (using just the basic name) and so its exact size is known. This value is made available in the $message_size variable. An example of how this can be used is described in the next section.

The filename used in maildir delivery should always be unique; however, in the unlikely case that such a file already exists, or if it fails to create the file, Exim waits for two seconds, and tries again with a new filename. The number of retries is controlled by the `maildir_retries` option (default 10). If this is exceeded, delivery is deferred.

Mailbox Quotas

If a system disk quota is exceeded while **appendfile** is writing to a file, the delivery is aborted and tried again later. A quota error is detectable by the retry rules (see Chapter 12), so the configuration file can specify special handling of these errors if necessary. After a quota error, **appendfile** does its best to clean up a partial delivery; any temporary files are deleted. If it is appending to a mailbox file, it resets the length and the time of last access to what they were before.

In some configurations, it may not be possible to make use of system quotas. To allow the sizes of mailboxes to be controlled in these cases, **appendfile** has its own quota mechanism.* The `quota` option imposes a limit on the size of the file to which Exim is appending, or to the total space used in the directory tree if the `directory` option is set. In the latter case, computation of the space used is expensive, because all the files in the directory (and any subdirectories) have to be individually inspected and their sizes summed up.† Also, there is no interlock against two simultaneous deliveries. It is preferable to use the quota mechanism in the operating system if you can.

* Of course, if system quotas are also in force, you cannot specify a quota that exceeds a system quota.

† This is done using the system's `stat()` function.

The cost of adding up the sizes of individual files can be lessened on systems where maildir delivery is in use and the users have no direct access to the files. Setting the following:

```
maildir_tag = ,S=$message_size
```

causes the size of each message to be added to its name, leading to message files with names such as:

```
/home/caesar/maildir/new/955429480.9324.host.example,S=3265
```

When Exim needs to find the size of a file, it first checks `quota_size_regex`. If this is set to a regular expression that matches the filename and it captures one string (by means of a parenthesized subexpression), that string is interpreted as a representation of the file's size. For example:

```
quota_size_regex = S=(\d+)$
```

could be used with the `maildir_tag` setting that was previously mentioned. This approach is not safe if the users have any kind of access that permits them to rename the files, but in environments where this is not the case, it saves having to run `stat()` for each file, which is of considerable benefit. The value of `quota_size_regex` is not expanded.

The value of the `quota` option itself is expanded, and the result must be a numerical value (decimal point allowed), optionally followed by one of the letters K or M. Thus, this sets a fixed quota of 10 MB for all users:

```
quota = 10M
```

The expansion happens while Exim is running as root or the Exim user, before it changes uid and gid in order to run the delivery, so files or databases that are inaccessible to the end user can be used to hold quota values that are looked up in the expansion. For example:

```
quota = pgsql;select quota from users \
        where id='${quote_pgsql:$local_part}'
```

When delivery fails because this quota is exceeded, the handling of the error is exactly as for system quota failures, and so it can be subject to special retry rules. The value specified is not accurate to the last byte, because the check is done before actually delivering the message. During delivery, separator lines and additional header lines may be added, thereby increasing the message's size by a small amount.

When messages are being delivered into individual files, the total number of files in the directory can also be controlled by setting the value of `quota_filecount` greater than zero. This can be used only if `quota` is also set.

Inclusive and Exclusive Quotas

As described so far, Exim's quota system operates in a similar way to system disk quotas: it prevents the mailbox from ever exceeding a certain size. This means that when a mailbox is nearly full, it may be possible to deliver small messages into it, but not large ones. Some administrators do not like this way of working; they would rather have a hard cutoff of all mail delivery when a quota is reached. In Exim 3.20, a new option was added to allow for this. It is called `quota_is_inclusive`, and by default it is set to true, which retains the default behavior. If you change the default by setting:

```
quota_is_inclusive = false
```

the check for exceeding the quota does not include the current message. Thus, deliveries continue until the quota has been exceeded; thereafter, no futher messages are delivered.

Quota Warnings

Users are often unaware of how large their mailboxes are getting, particularly when attachments have been sent to them. In the case of a single file, allowing a mailbox to exceed a few megabytes often causes the user's MUA to run slowly when initializing or when removing messages. This effect is often incorrectly reported as a slow-running system. The **appendfile** transport can be configured to send a warning message when the size of a mailbox crosses a given threshold. It does this only once, because repeating such a warning for mailboxes that are above the threshold would only exacerbate the problem.

The `quota_warn_threshold` option is expanded in the same way as `quota`. If the resulting value is greater than zero, and a successful delivery of the current message causes the size of the file or total space in the directory tree to cross the given threshold, a warning message is sent. It is not necessary to set `quota` in order to use this option,* but if it is set, the threshold may be specified as a percentage of the quota by following the value with a percent sign. For example:

```
quota = 10M
quota_warn_threshold = 75%
```

The warning message itself is specified by the `quota_warn_message` option, which must start with a *To:* header line containing the recipient(s). A *Subject:* line should also normally be supplied. The default is:

```
quota_warn_message = "\
  To: $local_part@$domain\n\
  Subject: Your mailbox\n\n\
```

* It can therefore be used with system quotas.

```
    This message is automatically created \
    by mail delivery software.\n\n\
    The size of your mailbox has exceeded \
    a warning threshold that is\n\
    set by the system administrator.\n"
```

Note the use of quotes to ensure that the escape sequence `\n` is recognized and turned into a newline character.

Notifying comsat

comsat is a server process that listens for reports of incoming mail and notifies logged-on users who have requested to be told when mail arrives by writing "You have mail" messages to their terminals. If `notify_comsat` is set, **appendfile** informs *comsat* when a successful delivery has been made. It is set in the default configuration file.

Summary of appendfile Options

The options that are specific to the **appendfile** transport are summarized in this section. Other options that can be set for **appendfile** are described earlier in the section "Summary of Generic Transport Options," the section "Environment for Local Transports," and the section "Options Common to the appendfile and pipe Transports."

`allow_fifo` (Boolean, default = false)
: If you want to deliver messages to FIFOs (named pipes), you must set this option to true, because **appendfile** will not deliver to a FIFO (named pipe) by default. If no process is reading the named pipe at delivery time, the delivery is deferred.

`allow_symlink` (Boolean, default = false)
: If you want to deliver messages to files using symbolic links, you must set this option true, because **appendfile** will not deliver to such files by default.

`check_group` (Boolean, default = false)
: The group owner of the file is checked to see that it is the same as the group under which the delivery process is running when this option is set. The default setting is unset because the default file mode is 0600, which means that the group is irrelevant.

`check_owner` (Boolean, default = true)
: If this option is turned off, the ownership of an existing mailbox file is not checked.

`create_directory` (Boolean, default = true)
: When this option is true, Exim creates any missing parent directories for the file that it is about to write. A created directory's mode is given by the `directory_mode` option.

`create_file` (string, default = `anywhere`)
: This option constrains the location of files that are created by the transport. It must be set to one of the value `anywhere`, `inhome`, or `belowhome`.

`directory` (string, default = unset)
: This option is mutually exclusive with the `file` option. When it is set, the string is expanded, and the message is delivered into a new file or files in or below the given directory, instead of being appended to a single mailbox file.

`directory_mode` (octal integer, default = 0700)
: If **appendfile** creates any directories as a result of the `create_directory` option, the mode is specified by this option.

`file` (string, default = unset)
: This option is mutually exclusive with the `directory` option. It need not be set when **appendfile** is being used to deliver to files whose names are obtained from forwarding, filtering, or aliasing address expansions, since in those cases the filename is associated with the address. Otherwise, either the `file` option or the `directory` option must be set.

`file_format` (string, default = unset)
: This option requests the transport to check the format of an existing file before adding to it. The check consists of matching a specific string at the start of the file.

`file_must_exist` (Boolean, default = false)
: If this option is true, the file specified by the `file` option must exist, and an error occurs if it does not. Otherwise, it is created if it does not exist.

`lock_fcntl_timeout` (time, default = `0s`)
: If this option is set to a nonzero time, blocking calls to `fcntl()` with that timeout are used to lock mailbox files. Otherwise nonblocking calls with sleeps and retries are used.

`lock_interval` (time, default = `3s`)
: This specifies the time to wait between attempts to lock the file.

`lock_retries` (integer, default = 10)
: This specifies the maximum number of attempts to lock the file. A value of zero is treated as 1.

`lockfile_mode` (octal integer, default = 0600)
This specifies the mode of the created lock file, when a lock file is being used.

`lockfile_timeout` (time, default = `30m`)
When a lock file is being used, if a lock file already exists and is older than this value, it is assumed to have been left behind by accident, and Exim attempts to remove it.

`maildir_format` (Boolean, default = false)
If this option is set with the `directory` option, delivery is into a new file in the maildir format that is used by some other mail software.

`maildir_retries` (integer, default = 10)
This option specifies the number of times to retry when writing a file in maildir format.

`maildir_tag` (string, default = unset)
This option applies only to deliveries in maildir format. It is expanded and added onto the names of newly delivered message files.

`mbx_format` (Boolean, default = false)
If `mbx_format` is set with the `file` option, the message is appended to the mailbox file in MBX format instead of traditional Berkeley Unix format. If none of the locking options are mentioned in the configuration, `use_mbx_lock` is assumed and the other locking options default to false.

`mode` (octal integer, default = 0600)
If a mailbox file is created, it is given this mode. If it already exists and has wider permissions, they are reduced to this mode. If it has narrower permissions, an error occurs unless `mode_fail_narrower` is false. However, if the delivery is the result of a `save` command in a filter file specifying a particular mode, the mode of the output file is always forced to take that value, and this option is ignored.

`mode_fail_narrower` (Boolean, default = true)
This option applies when an existing mailbox file has a narrower mode than that specified by the `mode` option. If `mode_fail_narrower` is true, the delivery is frozen ("mailbox has the wrong mode"); otherwise Exim continues with the delivery attempt, using the existing mode of the file.

`notify_comsat` (Boolean, default = false)
If this option is true, the *comsat* daemon is notified after every successful delivery to a user mailbox. This is the daemon that notifies logged-on users about incoming mail.

`quota` (string, default = unset)
This option imposes a limit on the size of the file to which Exim is appending, or to the total space used in the directory tree if the `directory` option is set. After expansion, the string must be numeric, optionally followed by K or M.

`quota_filecount` (integer, default = 0)
This option applies when the `directory` option is set. It limits the total number of files in the directory (like the inode limit in system quotas). It can only be used if `quota` is also set. A value of zero specifies no limit.

`quota_is_inclusive` (Boolean, default = true)
This option controls whether the current message is included when checking whether a mailbox has exceeded its quota. If the value is false, the check does not include the current message. In this case, deliveries continue until the quota has been exceeded; thereafter, no futher messages are delivered.

`quota_size_regex` (string, default = unset)
This option is used when Exim is computing the amount of space used in a directory by adding up the sizes of all the message files therein. It is not expanded, but is interpreted as a regular expression that is applied to every file name. If it matches and captures one string, that string is interpreted as a textual representation of the file's size.

`quota_warn_threshold` (string, default = `0`)
This option is expanded in the same way as `quota`. If the resulting value is greater than zero, and delivery of the message causes the size of the file or total space in the directory tree to cross the given threshold, a warning message is sent. The content and recipients of the message are defined by `quota_warn_message`. If `quota` is also set, the threshold may be specified as a percentage of it by following the value with a percent sign.

`quota_warn_message` (string, default = see description)
This string is expanded and inserted at the start of warning messages that are generated as a result of the `quota_warn_threshold` setting. It should start with a *To:* header line, which defines the recipient of the message.

`require_lockfile` (Boolean, default = true)
When a lock file is being used and `require_lockfile` is true, a lock file must be created before delivery can proceed. If the option is not true, failure to create a lock file is not treated as an error.

`use_fcntl_lock` (Boolean, default = true)
This option controls the use of the `fcntl()` function to lock a file for exclusive use when a message is being appended.

`use_lockfile` (Boolean, default = true)

If this option is turned off, Exim does not attempt to create a lock file when appending to a file.

`use_mbx_lock` (boolean, default = see description)

Setting the option specifies that special MBX locking rules be used. It is set by default if `mbx_format` is set and none of the locking options are mentioned in the configuration. The locking rules are the same as are used by the c-client library that underlies Pine4 and the IMAP4 and POP daemons that come with it. The rules allow for shared access to the mailbox. However, this kind of locking does not work when the mailbox is NFS-mounted.

The pipe Transport

The **pipe** transport delivers a message by creating a pipe and a new process that runs a given program. The message is written to the pipe by the transport, and read from the other end of the pipe by the external program. There are many common uses for this mechanism. For example:

- An individual user can set up a pipe delivery from a *.forward* file in order to process incoming messages automatically, perhaps to sort them into different folders or to generate automatic replies.*
- Messages addressed to mailing lists can be piped to a list handling program such as Majordomo or Listman.
- Messages addressed to domains reached by other transport mechanisms (such as UUCP) can be piped to programs that implement such transports.
- Local deliveries can be done by passing messages to an external local delivery agent (such as *procmail*), instead of using **appendfile**.

Defining the Command to Run

When a pipe is set up by aliasing or forwarding, or from a filter file, the command to run is defined by that mechanism. For example, an alias file could contain the line:

```
majordomo: |/usr/local/mail/majordomo
```

which causes messages addressed to the local part *majordomo* to be passed to a process running the command: */usr/local/mail/majordomo*. As another example, a user's filter file could contain the command:

* Some of this functionality is also available in Exim filter files.

```
pipe /usr/bin/vacation
```

which passes the message to */usr/bin/vacation*. A suitable transport for use in these cases is the one in the default configuration:

```
address_pipe:
  transport = pipe
  ignore_status
  return_output
```

where no command is specified, because it comes with the address that is being delivered (the other options are explained later in this chapter).

If the command name is not an absolute path, Exim looks for it in the directories listed by the colon-separated `path` option, whose default setting is:

```
path = /usr/bin
```

Thus, the previous filter example could actually be given as:

```
pipe vacation
```

If a router or director passes a message to a **pipe** transport directly, without involving aliasing or forwarding, the command is specified by the `command` option on the transport itself. For example, if you want your host to do all local deliveries using *procmail*, you can set up a transport like this:

```
procmail_pipe:
  driver = pipe
  command = /opt/local/bin/procmail -d $local_part
  return_path_add
  delivery_date_add
  envelope_to_add
  check_string = "From "
  escape_string = ">From "
  user = $local_part
  group = mail
```

The **localuser** director could be modified to use this transport like this:

```
procmail:
  driver = localuser
  transport = procmail_pipe
```

In this example, the pipe is run as the local user, but with the group set to *mail*. An alternative is to run the pipe as a specific user such as *mail* or *exim*; however, in this case you must arrange for *procmail* to trust that user to supply a correct sender address. If you do not specify either a `group` or a `user` option, the pipe command is run as the local user. The home directory is the user's home directory by default.

The Uid and Gid for the Command

The process that **pipe** sets up for its command runs under the same uid and gid as the **pipe** transport itself. As is the case for any local delivery, this user and group can be specified by the `user` and `group` options, either on the transport or on the director that calls it, or it can be taken from a local user's password file entry by the **localuser** or **forwardfile** directors.

Running the Command

Exim does not by default run the command for a **pipe** transport under a shell. This has two benefits:

- The additional cost of a shell process is avoided.
- Characters inserted into the command from the incoming message cannot be misinterpreted as shell metacharacters.

In the documentation for programs that expect to be run from an MTA via a pipe (for example, *procmail*), you often find a recommendation to place:

```
IFS=" "
```

at the start of the command. This assumes that the command will be run under a shell, and it is ensuring that the `IFS` variable (which defines the argument separator) is set to a space. When using the **pipe** transport in its default mode (that is, without using a shell, not only is this setting not required, but it will cause the command to fail because no shell is used.

Parsing the command line

String expansion is applied to the command line except when it comes from a traditional *.forward* file (commands from a filter file are expanded). However, before the expansion is done, the command line is broken down into a command name and a list of arguments. Unquoted arguments are delimited by whitespace; in double-quoted arguments, a backslash is interpreted as an escape character in the usual way. This does not happen for single-quoted arguments.

The expansion is applied to the command name, and to each argument in turn rather than to the whole line. Because the command name and arguments are identified before string expansion, any expansion item that contains whitespace must be quoted, so as to be contained within a single argument. A setting such as:

```
command = /some/path ${if eq{$local_part}{ab123}{xxx}{yyy}}
```

will not work, because it is split into the three items:

```
/some/path
${if
eq{$local_part}{ab123}{xxx}{yyy}}
```

and the second and third are not valid expansion items. You have to write:

```
command = /some/path "${if eq{$local_part}{ab123}{xxx}{yyy}}"
```

to ensure that the expansion is all in one argument. The expansion is done in this way, argument by argument, so that the number of arguments cannot be changed as a result of expansion, and quotes or backslashes in inserted variables do not interact with external quoting.

Special handling takes place when an argument consists precisely of the text `$pipe_addresses`. This is not a general expansion variable; the only place this string is recognized is when it appears as an argument for a pipe or transport filter command. It causes each address that is being handled to be inserted in the argument list *as a separate argument*. This makes it easy for the command to process the individual addresses, and avoids any problems with spaces or shell metacharacters. It is of use when a **pipe** transport is handling groups of addresses in a batch (see the `batch` option in the section "Batched Delivery and BSMTP," earlier in this chapter).

Using a shell

If a shell is needed in order to run the command, it can of course be explicitly specified. There are also circumstances where existing commands (for example, in existing *.forward* files) expect to be run under a shell and cannot easily be modified. To allow for these cases, there is an option called `use_shell`, which changes the way the **pipe** transport works. Instead of breaking up the command line as just described, it expands it as a single string and passes the result to */bin/sh*. This mechanism is inherently less secure, and in addition uses an extra process.

The Command Environment

The message that is being delivered is supplied to the command on its standard input stream, and the standard output and standard error streams are both connected to a single pipe that is read by Exim. The handling of output is described later in this section. The environment variables that are set up when the command is invoked are shown in Table 9-1.

Table 9-1. Environment Variables for Pipe Commands

Environment Variable	Meaning
`DOMAIN`	The local domain of the address
`HOME`	The "home" directory
`HOST`	The hostname when called from a router
`LOCAL_PART`	See the description later in this section
`LOGNAME`	See the description later in this section
`MESSAGE_ID`	The message's ID
`PATH`	As specified by the `path` option
`QUALIFY_DOMAIN`	The configured qualification domain
`SENDER`	The sender of the message
`SHELL`	`/bin/sh`
`USER`	See the description later in this section

The `environment` option can be used to add additional variables to this environment; its value is a colon-separated list of *`name`*`=`*`value`* settings, for example:

```
environment = PREFIX=$local_part_prefix
```

When a **pipe** transport is called directly from a director, `LOCAL_PART` is set to the local part of the address that the director handled. When it is called as a result of a forward or alias operation, `LOCAL_PART` is set to the local part of the address that was accepted by the aliasing or forwarding director. `LOGNAME` and `USER` are set to the same value as `LOCAL_PART` for compatibility with other MTAs.

`HOST` is set only when a **pipe** transport is called from a router as a pseudoremote transport (for example, for handling batched SMTP). It is set to the first hostname specified by the router (if any).

If the transport's `home_directory` option is set, its value is used for the `HOME` environment variable. Otherwise, any value that was set by the director, as described in the section "Adding Data for Use by Transports," in Chapter 7, is used.

The file creation mask (umask) setting in the pipe process is taken from the value of the `umask` option, which defaults to the value 022. This value means that any files that the process creates do not have the group- or world-writable permission bits set.

Timing the Command

A default timeout of one hour is imposed on the process that runs the command. If the command fails to complete within this time, it is killed. This normally causes the delivery to fail. The value of the timeout can be changed by the `timeout` option. A zero time interval specifies no timeout, but this is not recommended. In order to ensure that any further processes created by the command are also killed, Exim makes the initial process a process group leader, and kills the whole process group on a timeout. However, this action is undermined if any of the processes starts a new process group.

Restricting Which Commands Can Be Run

When users are permitted to set up pipe commands from *.forward* or filter files, you may want to restrict which commands they may specify. The `allow_commands` and `restrict_to_path` options provide two different ways of doing this. If neither are set, there is no restriction on which commands may be executed; otherwise, only commands that are permitted by one or the other of these options are allowed.

The `allow_commands` value is expanded, and then interpreted as a colon-separated list of permitted command names. They need not be absolute paths; the `path` option is used to resolve relative paths. If `restrict_to_path` is set, any command name not listed in `allow_commands` must contain no slashes (that is, it must be a simple command name); it is searched for only in the directories listed in the `path` option. For example, if the following is presented:

```
allow_commands = /usr/ucb/vacation
```

and `restrict_to_path` is not set, the only permitted command is */usr/ucb/vacation*. If, however, the configuration is:

```
allow_commands = /usr/ucb/vacation
path = /usr/local/bin
restrict_to_path
```

then in addition to */usr/ucb/vacation*, any command from */usr/local/bin* may be specified.

Enforcing the restrictions specified by `allow_commands` and `restrict_to_path` can be done only when the command is run directly from the **pipe** transport, without an intervening shell. Consequently, these options may not be set if `use_shell` is set.

Handling Command Errors

If `ignore_status` is true, the status (exit code) returned by the process that runs the command is ignored, and Exim always behaves as if zero (success) had been returned. If `ignore_status` is false (the default value), the status returned by the process can indicate a temporary or a permanent failure.

Temporary failures are the values listed in the `temp_errors` option. This contains a colon-separated list of numbers; the default contains the values of the errors

`EX_TEMPFAIL` and `EX_CANTCREAT`, which are commonly defined in */usr/include/sysexits.h*. After one of these errors, Exim defers delivery and tries again later.

Any other status value is treated as a permanent error, and the address is bounced. Failure to execute the command in a **pipe** transport is by default treated as a permanent failure. The most common causes of this are nonexistent commands and commands that cannot be accessed because of their permission settings. However, if `freeze_exec_fail` is set, failure to execute is treated specially, and causes the message to be frozen, whatever the setting of `ignore_status`.

Handling Output from the Command

Anything that the command writes to its standard error or standard output streams is "caught" by Exim. The maximum amount of output that the command may produce is limited by `max_output` (default 20 KB), as a guard against runaway programs. If the limit is exceeded, the process running the command is killed. This normally causes a delivery failure. Because of buffering effects, the amount of output may exceed the limit by a small amount before Exim notices.

You can control what happens to the output by setting a number of options. Three options set conditions under which the first line of any output is written to Exim's main log:

`log_defer_output`	log if delivery deferred
`log_fail_output`	log if delivery failed
`log_output`	log always

The output is converted to a single string of printing characters before being written to the log, using escape character sequences as necessary, in order not to disturb the layout of the log.

You can also arrange for the output to be returned to the sender of the message as part of a delivery failure report. If you set `return_output` true, the production of any output whatsoever is treated as contituting a delivery failure, independently of the return code from the command. The `return_fail_output` option operates in the same way, but applies only when the command process returns a nontemporary error code (that is, the return code is nonzero and is not one of those listed in `temp_errors`).

These options apply only when the message has a nonempty sender (that is, when it is not itself a bounce message). If neither of them is set, the output is discarded (after optional logging). However, even in this configuration, if the amount of output exceeds `max_output`, the command is killed on the grounds that is it probably misbehaving.

Summary of pipe Options

The options that are specific to the **pipe** transport are summarized in this section. Other options that can be set for **pipe** are described in the section "Summary of Generic Transport Options," the section "Environment for Local Transports," and the section "Options Common to the appendfile and pipe Transports," earlier in this chapter.

`allow_commands` (string, default = unset)
: The string is expanded, and then is interpreted as a colon-separated list of permitted commands. If `restrict_to_path` is not set, the only commands permitted are those in the `allow_commands` list. They need not be absolute paths; the `path` option is used for relative paths.

`command` (string, default = unset)
: This option need not be set when **pipe** is being used to deliver to pipes obtained from address expansions (usually under the instance name `address_pipe`). In other cases, the option must be set, to provide a command to run. It need not yield an absolute path (see the `path` option).

`environment` (string, default = unset)
: This option is used to add additional variables to the environment in which the command runs. Its value is a string that is expanded, and then interpreted as a colon-separated list of environment settings of the form *`name`*`=`*`value`*.

`freeze_exec_fail` (Boolean, default = false)
: Failure to execute the command in a **pipe** transport is by default treated like any other failure while running the command. However, if `freeze_exec_fail` is set, failure to execute is treated specially, and causes the message to be frozen, whatever the setting of `ignore_status`.

`ignore_status` (Boolean, default = false)
: If this option is true, the status returned by the process that is set up to run the command is ignored, and Exim behaves as if zero had been returned.

`log_defer_output` (Boolean, default = false)
: If this option is set and the status returned by the command is one of those listed in `temp_errors`, and any output was produced, the first line of it is written to the main log.

`log_fail_output` (Boolean, default = false)
: If this option is set, and the command returns any output, as well as terminates with a return code that is neither zero nor one of those listed in `temp_errors`, the first line of output is written to the main log.

`log_output` (Boolean, default = false)
: If this option is set and the command returns any output, the first line of output is written to the main log, whatever the return code.

`max_output` (integer, default = 20K)
: This specifies the maximum amount of output that the command may produce on its standard output and standard error file combined. If the limit is exceeded, the process running the command is killed.

`path` (string list, default = `/usr/bin`)
: This option specifies the string that is set up in the `PATH` environment variable of the subprocess. If the `command` option does not yield an absolute pathname, the command is sought in the *PATH* directories.

`restrict_to_path` (Boolean, default = false)
: When this option is set, any command name not listed in `allow_commands` must contain no slashes. The command is sought only in the directories listed in the `path` option.

`return_fail_output` (Boolean, default = false)
: If this option is true, and the command produces any output, as well as terminates with a return code other than zero or one of those listed in `temp_errors`, the output is returned in the delivery error message. However, if the message has a null sender (that is, it is itself a delivery error message), output from the command is discarded.

`return_output` (Boolean, default = false)
: If this option is true, and the command produces any output, the delivery is deemed to have failed whatever the return code from the command, and the output is returned in the delivery error message. Otherwise, the output is just discarded. However, if the message has a null sender (that is, it is a delivery error message), output from the command is always discarded, whatever the setting of this option.

`temp_errors` (string, default = see description)
: This option contains a colon-separated list of numbers. If `ignore_status` is false and the command exits with a return code that matches one of the numbers, the failure is treated as temporary and the delivery is deferred. The default setting contains the codes defined by `EX_TEMPFAIL` and `EX_CANTCREAT` in *sysexits.h*. If Exim is compiled on a system that does not define these macros, it assumes values of 75 and 73, respectively.

`timeout` (time, default = `1h`)
: If the command fails to complete within this time, it is killed. This normally causes the delivery to fail. A zero time interval specifies no timeout.

`umask` (octal integer, default = 022)
This specifies the umask setting for the process that runs the command.

`use_shell` (Boolean, default = false)
If this option is set, it causes the command to be passed to */bin/sh* instead of being run directly from the transport. This is less secure, but is needed in some situations where the command is expected to be run under a shell and cannot easily be modified. You cannot make use of the `allow_commands` and `restrict_to_path` options, or the $pipe_addresses facility if you set `use_shell` true. The command is expanded as a single string, and handed to */bin/sh* as data for its *-c* option.

The lmtp Transport

When an installation supports a very large number of accounts, keeping individual mailboxes as separate files or directories is no longer feasible because of the problems of scale. One solution to this problem is to use an independent *message store*, which is a software product for managing mailboxes. It provides interfaces for adding, reading, and deleting messages, but how they are stored internally is not defined, and the mailboxes cannot be accessed as regular files. One such product is the Cyrus IMAP message store.*

When an MTA delivers a message to a message store of this type, it has to pass over the envelope just as it does when delivering to a remote host, and the existing SMTP protocol seems like a good candidate for a means of doing this. However, for messages with multiple recipients, there are some technical problems in using SMTP in this way, which led to the definition of a new protocol called LMTP.† This is very similar to SMTP, and Exim's **smtp** transport has an option for using LMTP instead of SMTP over a TCP/IP connection (see the section "Use of the LMTP Protocol," earlier in this chapter).

LMTP is also intended for communication between two processes running on the same host, and this is where the **lmtp** transport comes in.‡ It is in effect a cross between the **pipe** and **smtp** transports. Like **pipe**, it runs a command in a new process and sends the message to it using a pipe, but instead of just writing the message down the pipe, it interacts with the command to operate the LMTP protocol. Here is an example of a typical **lmtp** transport:

* See *http://asg.web.cmu.edu/cyrus*.

† If you are interested in the detailed arguments, read RFC 2033, which defines LMTP.

‡ Because LMTP is not required in the majority of installations, the code for the **lmtp** transport is not included in Exim unless specially requested at build time.

```
local_lmtp:
  driver = lmtp
  command = /some/local/lmtp/delivery/program
  batch = all
  batch_max = 20
  user = exim
```

This delivers up to 20 addresses at time, in a mixture of domains if necessary, running as the user *exim*.

The **lmtp** transport has a small number of options, all of which operate in exactly the same way as the **pipe** options of the same names, so rather than repeat the discussion, we just list them in Table 9-2.

Table 9-2. Private Options for the lmtp Transport

Option	Meaning
`batch`	Allow multiple addresses to be batched
`batch_max`	Limit the number of batched addresses
`command`	Define the command to be run
`group`	The group under which to run the command
`timeout`	Maximum time to wait for a response
`user`	The user under which to run the command

Since the whole point of LMTP is to be able to pass a single copy of a message with more than one recipient, `batch` should normally be set to a value other than the default. As for other local transports, if `user` or `group` is not set, values must be set up by the director that passes addresses to this transport.

The autoreply Transport

The **autoreply** transport is not a true transport in that it does not cause the message to be delivered in the usual sense. Instead, it generates another, new mail message. However, the original message can be included in the generated message.

This transport is commonly run as the result of mail filtering, where sending a "vacation" message is a common example. What the user wants to do is to send an automatic response to incoming mail, saying that he or she is away and therefore will not be able to read the mail for a while. In the next chapter, we go into mail filtering in depth, but here is an extract from a filter file that does this job:

```
if personal and not error_message then
   mail
   to $reply_address
   subject "Re: $h_subject"
```

```
    text "I'm away this week, but I'll get back to you asap."
    endif
```

In an attempt to reduce the possibility of message cascades, messages created by the **autoreply** transport always take the form of delivery error messages. That is, the envelope sender field is empty. This should stop hosts that receive them from generating new automatic responses in turn.

The **autoreply** transport is implemented as a local transport so that, when activated from a user's filter file, it runs under the uid and gid of the local user and with appropriate current and home directories.

There is a subtle difference between directing a message to a **pipe** transport that generates some text to be returned to the sender, and directing it to an **autoreply** transport. This difference is noticeable only if more than one address from the same message is so handled. In the case of a **pipe**, the separate outputs from the different addresses are gathered up and returned to the sender in a single message, whereas if **autoreply** is used, a separate message is generated for each address that is passed to it.

If any of the generic options for manipulating headers (for example, `headers_add`) are set on an **autoreply** transport, they apply only to the copy of the original message that is included in the generated message when `return_message` is set. They do not apply to the generated message itself, and are therefore not really useful.

If the **autoreply** transport receives return code 2 from Exim when it submits the new message, indicating that there were no recipients, it does not treat this as an error. This means that autoreplies that are addressed to $sender_address when this is empty (because the incoming message is a delivery failure report) do not cause problems.

The Parameters of the Message

There are two possible sources of the data for constructing the new message: the incoming address and the options of the transport. When a user's filter file contains a *mail* or *vacation* command, all the data from the command (recipients, header lines, body) is attached to the address that is passed to the **autoreply** transport. In this case, the transport's options are ignored.

On the other hand, when the transport is activated directly by a director or router (that is, not from a filter file), the data for the message is taken from the transport's options. In other words, the options in the transport's configuration are used only when it receives an address that does not contain any reply information of its own. Thus, the message is specified entirely by the filter file or entirely by the transport; it is never built from a mixture of data.

When the data has not come from a filter command, the transport options that specify the message are shown in the following list:

`bcc`, `cc`, `to`
: Specifies recipients and the corresponding header lines.

`from`
: Specifies the *From:* header line; if this does not correspond to the user running the transport, a *Sender:* header is added by Exim.*

`reply_to`
: Sets the *Reply-To:* header line. Note that the option name contains an underscore, not a hyphen.

`subject`
: Sets the *Subject:* header line.

`text`
: Sets a short text to appear at the start of the body.

`file`
: Specifies a file whose contents form the body of the message. If `file_optional` is set, no error is generated if the file does not exist or cannot be read. If `file_expand` is set, the contents of the file are passed through the string expander, line by line, as they are added to the message. This makes it possible to vary the contents of the message according to the circumstances.

`headers`
: Specifies additional header lines that are to be added to the message. Several headers can be added by enclosing the text in quotes and using `\n` to separate them.

`return_message`
: Specifies that the original message is to be added to the end of the newly created message. The amount that is returned is subject to the general `return_size_limit` option.

For example, here is a transport that could be used to send back a message explaining that a particular mailing list no longer exists:

```
auto_message:
  driver = autoreply
  to = $sender_address
  subject = The mailing list $local_part is no more
  text = "Your message to $local_part@$domain is being returned because\n\
    the $local_part mailing list is no longer in use."
  return_message
```

* See the `local_from_check` and `local_from_prefix` options in the section "Sender Address," in Chapter 13, *Message Reception and Policy Controls*, for ways to change this behavior.

You could direct messages to this transport by a director such as this:

```
old_lists:
  driver = smartuser
  domains = mailing.list.domain
  local_parts = /etc/dead/lists
  transport = auto_message
```

This runs only for the one domain, for local parts that are listed in the file.

Once-Only Messages

The traditional "vacation" use of **autoreply** is to send a message only once, or no more than once in a certain time interval, to each different sender. This can be configured by setting the `once` option to the name of a file, which is then used to record the recipients of messages that are created, together with the times at which the messages are sent. By default, only one message is ever sent to a given recipient. However, if `once_repeat` is set to a time greater than zero, another message may be sent if that much time has elapsed since the previous message. For example:

```
once_repeat = 10d
```

The settings of `once` and `once_repeat` are used only if the data for the message is being taken from the transport's own options. For calls of **autoreply** that originate in message filters, the settings from the filter are used.

Keeping a Log of Messages Sent

The `log` option names a file in which a record of every message that is handled by the transport is logged. The value of the option is expanded, and the file is created if necessary, with a mode specified by `mode`.

The setting of `log` is used only if the data for the message is being taken from the transport's own options. For calls of **autoreply** that originate in message filters, the settings from the filter are used.

Summary of autoreply Options

The options that are specific to the **autoreply** transport are summarized in this section. Other options that can be set for **autoreply** are described earlier in the section "Summary of Generic Transport Options," and the section "Environment for Local Transports."

`bcc` (string, default = unset)
: Specifies the addresses that are to receive "blind carbon copies" of the message when the message is specified by the transport. The string is expanded.

`cc` (string, default = unset)
Specifies recipients of the message and the contents of the *Cc:* header when the message is specified by the transport. The string is expanded.

`file` (string, default = unset)
The contents of the file are sent as the body of the message when the message is specified by the transport. The string is expanded. If both `file` and `text` are set, the text string comes first.

`file_expand` (Boolean, default = false)
If this is set, the contents of the file named by the `file` option are subjected to string expansion as they are added to the message.

`file_optional` (Boolean, default = false)
If this option is true, no error is generated if the file named by the `file` option does not exist or cannot be read.

`from` (string, default = unset)
The contents of the *From:* header when the message is specified by the transport. The string is expanded.

`headers` (string, default = unset)
Specified additional RFC 822 headers that are to be added to the message when the message is specified by the transport. The string is expanded.

`log` (string, default = unset)
This option names a file in which a record of every message sent is logged when the message is specified by the transport. The string is expanded.

`mode` (octal integer, default = 0600)
If either the log file or the "once" file has to be created, this mode is used.

`once` (string, default = unset)
This option names a DBM database in which a record of each recipient is kept when the message is specified by the transport. The string is expanded.

`once_repeat` (time, default = `0s`)
This option specifies the time interval after which another message may be sent to the same recipient, when the message is specified by the transport.

`reply_to` (string, default = unset)
This option specifies the contents of the *Reply-To:* header when the message is specified by the transport. The string is expanded.

`return_message` (Boolean, default = false)
If this is set, a copy of the original message is returned with the new message, subject to the maximum size set in the `return_size_limit` general configuration option.

`subject` (string, default = unset)
The contents of the *Subject:* header when the message is specified by the transport. The string is expanded.

`text` (string, default = unset)
This specifies a single string to be used as the body of the message when the message is specified by the transport. The string is expanded. If both `text` and `file` are set, the text comes first.

`to` (string, default = unset)
This option specifies recipients of the message and the contents of the *To:* header when the message is specified by the transport. The string is expanded.

10

Message Filtering

The word *filtering* is used for the process of inspecting a message as it passes through Exim, and possibly changing the way it is handled. We discuss transport filters in the section "Transport Filters," in Chapter 9, *The Transports*. In this chapter, we are concerned with a different kind of filtering, which happens while a message is being processed to determine where it should be delivered. We have already mentioned user and system filters without much explanation; now is the time to rectify that omission. These filters work as follows:

- Provided the configuration permits it, users may place filtering instructions in their *.forward* files instead of just a list of forwarding destinations.[*] User filters are obeyed when Exim is directing addresses, and they extend the concept of forwarding by allowing conditions to be tested. A user filter is run as a consequence of directing one address, whose constituent parts are available in $local_part and $domain (and, if relevant, $local_part_prefix and/or $local_part_suffix).
- A single system filter can be set up by the administrator. This uses the same filtering commands as a user filter (with a few additions), but is obeyed just once per delivery attempt, before any directing or routing is done. Because there may be many recipients for a message, address-related variables such as $local_part and $domain are not set when a system filter is run, but a list of all the recipients is available in the variable $recipients.

Before describing the syntax of filtering commands in detail, we work through a few straightforward examples to give you a flavor of how filtering operates.

[*] The filename *.forward* is the one most commonly used, but it is not hardwired into Exim. The configuration could specify a different name.

Examples of Filter Commands

These examples are written from the perspective of a user's filter file. First, a simple forwarding:

```
# Exim filter
deliver baggins@rivendell.middle-earth.example
```

The first line indicates that the file is a filter file rather than a traditional *.forward* file. The only command in this file is an instruction to deliver the message to a specific address. This particular file does nothing that a traditional *.forward* couldn't do, and is exactly equivalent to a file containing:

```
baggins@rivendell.middle-earth.example
```

The next example shows vacation handling using traditional means; that is, by running `/usr/ucb/vacation`, assuming that *.vacation.msg* and other files have been set up in the home directory:

```
# Exim filter
unseen pipe "/usr/ucb/vacation $local_part"
```

The `pipe` command is an instruction to run the given program and pass the message to it via a pipe. The arguments given to filter commands are always expanded so that references to variables such as $local_part can be included.

The word `unseen`, which precedes the command, tells Exim not to treat this command as *significant.* This means that after filtering, Exim goes on to deliver the message in the normal way. That is, one copy of the message is sent to the pipe, and another is added to the user's mailbox. Without `unseen`, the pipe delivery would be the only delivery that is done.

This example also doesn't do anything that a traditional *.forward* couldn't do. For the user *spqr*, it is equivalent to:

```
\spqr, |/usr/ucb/vacation spqr
```

Vacation handling can, however, be done entirely inside an Exim filter, without running another program. Assuming there is a file called *.vacation.msg* in the home directory, this filter file suffices:

```
# Exim filter
if personal then vacation endif
```

Here we see something that a traditional *.forward* file cannot do. The *if* command allows a filter to test certain conditions before taking action. In this example, it is testing the `personal` condition. We explain what this means in detail later on, but for now you just need to know that it is distinguishing between messages that are personally addressed to the user, and those that are not (for example, messages from a mailing list). If the incoming message is a personal one, the filter command *vacation* is run. This command generates an automatic response message to the sender of the incoming message, in the same way that */usr/ucb/vacation* does.

As well as sending the vacation message, we want Exim to continue processing the message, and do the normal delivery. In this case, this happens automatically, because the *vacation* command is not a significant action. There is no need to use `unseen`, because it is the default.

There are a range of conditions that you can test with the *if* command. The next example examines the contents of the *Subject:* header line, and for certain subjects, arranges to deliver the message to a specific file instead of the normal mailbox:

```
# Exim filter
if $header_subject: contains "empire" or
   $header_subject: contains "foundation"
then
   save $home/mail/f+e
endif
```

Saving to a file like this is a significant action, so no other deliveries are done for messages that match the test. If the *save* command were preceded by `unseen`, it would take a copy of the relevant messages, without affecting normal delivery.

The following example illustrates the use of a regular expression for a slightly unusual purpose. It extracts the day of the week from the variable $tod_full, which contains the date and time in the format:

```
Wed, 16 Oct 1995 09:51:40 +0100
```

The filter command looks for messages that are not marked "urgent," and saves them in files whose names contain the day of the week:

```
# Exim filter
if $header_subject: does not contain "urgent" and
   $tod_full matches "^(...),"
then
  save $home/mail/$1
endif
```

The regular expression always matches, and it uses parentheses to extract the day name into the $1 variable. This can then be used in the command that follows.

Suppose you want to throw away all messages from a certain domain that is bombarding you with junk, but also want to accept messages from the postmaster. This filter file achieves this:

```
# Exim filter
if $reply_address contains "@spam.site.example" and
   $reply_address does not contain "postmaster@"
```

```
then
   seen finish
endif
```

The *finish* command ends filtering. Putting the word `seen` in front of it makes it into a significant action, which means that normal message delivery will not take place. As no deliveries are set up by the filter when the condition matches, the message is discarded.

This final example shows how a user can handle multiple personal mailboxes, as described in the section "Conditional Running of Directors," in Chapter 7, *The Directors*:

```
# Exim filter
if $local_part_suffix is "-foo"
then
  save $home/mail/foo
elif $local_part_suffix is "-bar"
then
  save $home/mail/bar
endif
```

The remainder of this chapter covers the format of filter files and the individual filtering commands in detail.

Filtering Compared with an External Delivery Agent

It is important to realize that no deliveries are actually done while a system or user filter file is being processed. The result of filtering is a list of destinations to which a message should be delivered; the deliveries themselves take place later, along with all other deliveries for the message. This means that it is not possible to test for successful deliveries while filtering. It also means that duplicate addresses generated by filtering are dropped, as with any other duplicate addresses.

If a user needs to be able to test the result of a delivery in some automatic way, an external delivery agent such as *procmail* must be used.* At first sight, *procmail* and Exim filters appear to provide much the same functionality, albeit with very different syntax. However, there are some important differences:

- Forwarding from an Exim filter is "true forwarding" in the sense that the envelope sender is not changed; the message is just redirected to a new recipient. An unprivileged external delivery agent cannot do this because any message it resubmits has the local user's address as its envelope sender.

* See the section "Using an External Local Delivery Agent," in Chapter 5, *Extending the Delivery Configuration*.

- Because an external delivery agent has to submit a new message in order to achieve additional deliveries, duplicate addresses are not detected. For example, if a message arrives addressed to both *bob* and *alice*, and *bob* forwards it to *alice* from *procmail*, two copies are delivered, whereas if the forwarding happens in an Exim filter, only one copy is delivered.
- As mentioned earlier, the results of a delivery (for example, the return code from the command to which a message is piped) can be inspected by an external delivery agent, but not in an Exim filter.

The use of Exim filters and external delivery agents is not mutually exclusive. You should use whichever of them best provides the features you need.

Setting Up a User Filter

If a user's *.forward* file begins with the text:

```
# Exim filter
```

in any capitalization and with any spacing, the file is interpreted as a filter file instead of a conventional forwarding list, provided that the administrator has enabled filtering by setting the `filter` option on the **forwardfile** director. The remaining contents of the filter file must conform to the filtering syntax, which is described later in this chapter. As in the case of a conventional *.forward* file, if the filter sets up any deliveries to pipes or files, the **forwardfile** director must have settings of `address_pipe_transport` or `address_file_transport` to specify how such deliveries are to be done (see the section "Use of Transports by Directors and Routers," in Chapter 3, *Exim Overview*).

Setting Up a System Filter

A system filter is different to a user filter in that it runs only once, however many recipients a message might have. Because it runs right at the start of a delivery process, before the recipient addresses are directed or routed, the options for setting it up appear in the main section of the runtime configuration file.

A system filter is enabled by setting the option `message_filter` to the path of the file that contains the filter instructions. For example:

```
message_filter = /etc/mail/exim.filter
```

If any commands in the filter specify deliveries to pipes or files, or the creation of automatic reply messages, additional options must be set to specify which transports are to be used for these purposes. These options are as follows:

`message_filter_directory_transport`	delivery to directory
`message_filter_file_transport`	delivery to file
`message_filter_pipe_transport`	delivery to pipe
`message_filter_reply_transport`	automatic reply

The *save* command in a filter specifies delivery to a given file or directory. If the path name does not end with a slash character, it is assumed to be the name of a file, and the transport specified by `message_filter_file_transport` is used; otherwise, delivery into a new file within a given directory is assumed, and `message_filter_directory_transport` is used.*

Two further options, `message_filter_user` and `message_filter_group`, specify the uid and gid under which the system filter is run. This is achieved by temporarily changing the effective uid and gid. If these options are not set, the uid and gid are not changed when the system filter is run. If the filter generates any pipe or file deliveries, or any automatic replies, the uid and gid under which the filter is run are used when running the appropriate transports, unless the transport configurations override them.

Summary of System Filter Options

The options that are concerned with setting up a system filter are summarized in this section:

`message_filter` (string, default = unset)
This option specifies the system filter file, and enables system filtering.

`message_filter_directory_transport` (string, default = unset)
This sets the name of the transport driver that is to be used when the *save* command in a system message filter specifies a path ending in `/`, implying delivery of each message into a separate file in some directory.

`message_filter_file_transport` (string, default = unset)
This sets the name of the transport driver that is to be used when the *save* command in a system message filter specifies a path not ending in `/`.

`message_filter_group` (string, default = unset)
This option sets the gid under which the system message filter is run. The same gid is used for any pipe, file, or autoreply deliveries that are set up by the filter, unless the transport overrides them.

* These options are just pointers to transports that would normally do the kind of delivery implied, but there is no constraint on what they actually do.

`message_filter_pipe_transport` (string, default = unset)
This sets the name of the transport driver that is to be used when a *pipe* command is used in a system message filter.

`message_filter_reply_transport` (string, default = unset)
This sets the name of the transport driver that is to be used when a *mail* command is used in a system message filter.

`message_filter_user` (string, default = unset)
This option sets the uid under which the system message filter is run. The same uid is used for any pipe, file, or autoreply deliveries that are set up by the filter, unless the transport overrides them.

Testing Filter Files

Filter files, especially the more complicated ones, should always be tested, as it is easy to make mistakes. Exim provides a facility for preliminary testing of a filter file before installing it. This tests the syntax of the file and its basic operation, and can also be used with ordinary (nonfilter) *.forward* files.

Because a filter can do tests on the content of messages, a test message is required. Suppose you have a new user filter file called *new-filter* and a test message in a file called *test-message*. The following command can be used to test the filter:

```
exim -bf new-filter <test-message
```

The `-bf` option tells Exim that the following item on the command line is the name of a filter file that is to be tested, and the test message is supplied on the standard input. If there are no message-dependent tests in the filter, an empty file can be used. A supplied message must start with header lines or the `From` message separator line that is found in traditional multimessage folder files. A warning is given if no header lines are read.

The result of running this command, provided no errors are detected in the filter file, is a list of the actions that Exim would try to take if presented with the message for real. For example, the output:

```
Deliver message to: gulliver@lilliput.example
Save message to: /home/lemuel/mail/archive
```

means that one copy of the message would be sent to *gulliver@lilliput.example*, and another would be added to the file */home/lemuel/mail/archive*, if all went well.

The actions themselves are not attempted while testing a filter file in this way; there is no check, for example, that any forwarding addresses are valid. If you want to know why a particular action is being taken, add the `-v` option to the command. This causes Exim to output the results of any conditional tests and to indent its output according to the depth of nesting of *if* commands in the filter file. Further additional output from a filter test can be generated by the *testprint* command, which is described later.

When Exim is outputting a list of the actions it would take, if any text strings are included in the output, nonprinting characters therein are converted to escape sequences. In particular, if any text string contains a newline character, this is shown as `\n` in the testing output.

When testing a filter, Exim makes up an envelope for the message. The recipient is by default the user running the command, and so is the sender, but the command can be run with the `-f` option to supply a different sender. For example:

```
exim -bf new-filter \
  -f islington@neverwhere.example <test-message
```

Alternatively, if the `-f` option is not used, but the first line of the supplied message is a `From` separator from a message folder file (not the same thing as a *From:* header line), the sender is taken from there. If `-f` is present, the contents of any `From` line are ignored.

The return path is the same as the envelope sender, unless the message contains a *Return-path:* header, in which case it is taken from there. You need not worry about any of this unless you want to test out features of a filter file that rely on the sender address or the return path.

It is possible to change the envelope recipient by specifying further options. The `-bfd` option changes the domain of the recipient address, while the `-bfl` option changes the local part. An adviser could make use of these options to test someone else's filter file, for example.

The `-bfp` and `-bfs` options specify the prefix or suffix for the local part. See the section "Multiple User Addresses," in Chapter 5 for an example of when these might be relevant.

If the filter tests information about the source of the message (for example, the name or the IP address of the host from which it was received), you may want to set up specific values for it to test. This can be done by making use of several command-line options, beginning with `-oM`. For example, `-oMa` sets the remote host address. See Chapter 20, *Command-Line Interface to Exim*, for details of these options.

Testing a System Filter File

A system filter can be tested in the same way as a user filter, but you should use the command-line option `-bF` instead of `-bf`. This allows Exim to recognize those commands and other features of a system filter that are not available in user filters.

Testing an Installed Filter File

Testing a filter file before installation cannot find every potential problem; for example, it does not actually run commands to which messages are piped. Some "live" tests should therefore also be done once a filter is installed.

If at all possible, users should test their filter files by sending messages from other accounts. If a user sends a test message from the filtered account and delivery fails, the error message is sent back to the same account, which may cause another delivery failure. It will not cause an infinite sequence of such messages, because delivery failure messages do not themselves generate further messages. However, it does mean that the failure will not be returned to the sender, and also that the postmaster will have to investigate the stuck message.

A sensible precaution against this occurrence is to include the line:

```
    if error_message then finish endif
```

as the first filter command, at least while testing. This causes filtering to be abandoned for a delivery failure message, and since no destinations are generated by the filter, the message goes on to be delivered to the original address. Unless there is a good reason for not doing so, it is recommended that the previous line be present at the start of all user filter files.

Format of Filter Files

Apart from leading whitespace, the first text in a filter file must be:

```
    # Exim filter
```

This is what distinguishes it from a conventional *.forward* file (assuming the configuration has enabled filtering). If the file does not have this initial line, it is treated as a conventional *.forward* file, both when delivering mail and when using the `-bf` testing mechanism. The whitespace in the line is optional, and any capitalization may be used. Further text on the same line is treated as a comment. For example, you could have:

```
    #   Exim filter   <<== do not edit or remove this line!
```

The remainder of the file is a sequence of filtering commands, which consist of keywords and data values separated by whitespace or line breaks, except in the case of conditions for the *if* command, where parentheses also act as separators.

For example, in the command:

```
deliver gulliver@lilliput.example
```

the keyword is `deliver` and the data value is `gulliver@lilliput.example`. The commands are in free format, and can be spread over more than one line; there are no special terminators. If the character `#` follows a separator, everything from `#` up to the next newline is ignored. This provides a way of including comments in a filter file.

There are two ways in which a data value can be input:

- If the text contains no whitespace, it can be typed verbatim. However, if it is part of a condition, it must also be free of parentheses, because these are used for grouping in conditions. The examples shown so far have all been of this type.
- Otherwise, a data value must be enclosed in double quotation marks, for example:

  ```
  if $h_subject: contains "Free Gift" then
    save /dev/null
  endif
  ```

When quotes are used, backslash is treated as an "escape character" within the string, thereby allowing special characters such as newline to be included. A data item enclosed in double quotes can be continued onto the next line by ending the first line with a backslash. Any leading whitespace at the start of the continuation line is ignored.

In addition to the escape character processing that occurs when strings are enclosed in quotes, most data values are also subject to string expansion. In an expanded string, both the dollar and backslash characters are interpreted specially. This means that if you really want a dollar in a data item in a filter file, you have to escape it. The command:

```
if $h_subject: contains \$\$\$\$ then seen finish endif
```

tests for the string `$$$$`. If quotes are used, an additional level of escaping is necessary:

```
if $h_subject: contains "\\$\\$\\$\\$" then
  seen finish
endif
```

If a backslash is required in a quoted data string, as can happen if the string is to be interpreted as a regular expression, `\\\\` has to be entered.

Significant Actions

When in the course of delivery a message is processed by a filter file, what happens next (that is, after the filter file has been processed), depends on whether the filter has taken any *significant action* or not. For a user filter, if there is at least one significant action, the filter is considered to have handled the entire delivery arrangements for the current address, and no further processing of the address takes place. In the case of a system filter, if any significant actions are taken, the original recipient addresses are ignored.

If, on the other hand, no significant actions happen, Exim continues processing as if there were no filter file. For a user filter, the address is offered to subsequent directors, and normally this eventually sets up delivery of a copy of the message into a local mailbox. For a system filter, the original recipient addresses are directed or routed, and delivery proceeds as normal.

The delivery commands *deliver*, *save*, and *pipe* are by default significant actions. For example, if the command:

```
deliver hatter@wonderland.example
```

is obeyed in a user's filter file, the address is not offered to subsequent directors. However, if the command is preceded by the word `unseen`, for example:

```
unseen deliver hatter@wonderland.example
```

the delivery is not considered to be significant. In effect, a filter containing only an "unseen" delivery takes a copy of a message, without affecting the normal delivery. This can be used in a system filter for archiving messages automatically.

The other filter commands (those that do not specify a delivery of the message) are not significant actions by default, but they can be made significant by putting the word `seen` before them. For example, obeying a *finish* command terminates the running of a filter; it is not by itself a significant action, so whether the filter as a whole has taken any significant action depends on the earlier commands (if any). However, if the command:

```
seen finish
```

is obeyed, the filter ends with a significant action, and no further delivery processing takes place. A filter containing only this command is a black hole; it is most commonly used after testing some characteristic of the message.

Filter Commands

The filter commands described in subsequent sections are as follows:

add	Increment a user variable
deliver	Deliver to an email address
fail	Fail delivery (system filter only)
finish	End processing
freeze	Freeze delivery (system filter only)
headers	Add/remove header lines (system filter only)
if	Test condition
logfile	Define log file
logwrite	Write to log file
mail	Send a reply message
pipe	Pipe to a command
save	Save to a file
testprint	Print while testing
vacation	Tailored form of *mail*

The add Command

```
add number to user variable
example: add 2 to n3
```

The names of the user variables consist of the letter `n` followed by a single digit. There are therefore 10 user variables of this type, and their values can be obtained by the normal expansion syntax (for example $n4) in other commands. At the start of filtering, these variables all contain zero. At the end of a system filter, their values are copied into $sn0 to $sn9 so that they can be referenced in users' filter files. Thus, a system filter can set up "scores" for a message, to which a user filter can refer. Both arguments of the *add* command are expanded before use, making it possible to add variables to each other. Subtraction can be obtained by adding negative numbers. The following example is not realistic, but shows the kind of thing that can be done:

```
if $h_subject: does not match "(?-i)[a-z]" then
  add 1 to n1
endif
if $h_subject: contains "make money" then
  add 1 to n2
endif
add $n2 to n1
if $n1 is above 3 then seen finish endif
```

The first command tests the subject of the message for the presence of a lowercase letter. If there are none (that is, if the subject is entirely in uppercase), the variable $n1 is incremented. The second command increments $n2 if the subject contains a specific string. Then the contents of $n2are added to $n1, and the sum is tested.

Notice that the variable names are given without a leading dollar for variables that are being incremented. If you wrote:

```
add 1 to $n1
```

the string expansion that is applied to each argument would turn it into something such as:

```
add 1 to 0
```

which is an invalid command that provokes a syntax error.

Delivery Commands

The following filter commands set up message deliveries (that is, they arrange for copies of the message to be transported somewhere). Such deliveries are significant actions unless the command is preceded by `unseen`.

The deliver Command

```
deliver mail address
example: deliver "Dr Livingstone <David@darkest.africa.example>"
```

This provides a forwarding operation. The message is sent to the given address. For a user filter, this is exactly the same as putting the address in a traditional *.forward* file. To deliver a copy of the message to a user's normal mailbox, the user's login name can be given. Once an address has been processed by the filtering mechanism, an identical generated address will not be so processed again, so doing this does not cause a loop.

An optional addition to a *deliver* command is:

```
errors_to mail address
```

In a system filter, the given address is not restricted, but in the case of a user filter, it must be the address that is in the process of being directed. That is, the only valid usage is:

```
errors_to $local_part@$domain
```

(or the equivalent using a literal string) in a user filter. This facility allows users to change the envelope sender of a message to be their own address when forwarding it, so that any subsequent delivery failure reports are sent to the forwarding user, instead of to the original sender. This is useful only when the forwarding is conditional, of course, so that bounce messages are not themselves being forwarded.

Only a single address may be given to a *deliver* command, but multiple occurrences of the command may be used to cause the message to be delivered to more than one address. However, duplicate addresses are discarded.

The save Command

```
save file name
example: save $home/mail/bookfolder
```

This causes a copy of the message to be appended to the given file (that is, the file is used as a mail folder). More than one *save* command may appear; each one causes a copy of the message to be written to its argument file, provided they are different (duplicate *save* commands are ignored).

The ability to use the *save* command in a user filter is controlled by the system administrator; it may be forbidden by setting `forbid_file` on the **forwardfile** director.

An optional mode value may be given after the filename, for example:

```
save /some/folder 0640
```

This makes it possible for users to override the systemwide mode setting for file deliveries, which is normally `0600`. If an existing file does not have the correct mode, it is changed. The value for the mode is interpreted as an octal number, even if it does not begin with a zero.

For a system filter, the filename must be an absolute path. For a user filter, if the filename does not start with a slash character, the directory specified by the $home variable is prepended. The user must of course have permission to write to the file, and (in a conventional configuration) the writing of the file takes place in a process that is running with the user's uid and the user's primary gid. Any secondary groups to which the user may belong are not normally taken into account, though the system administrator can configure Exim to set them up.*

An alternative form of delivery may be enabled, in which each message is delivered into a new file in a given directory. For a system filter, this requires a setting of `message_filter_directory_transport`, whereas for a user filter it requires a setting of `directory_transport` on the **forwardfile** director. If this is the case, the

* See the discussion of the `initgroups` option in the section "Controlling the Environment for Local Deliveries," in Chapter 6, *Options Common to Directors and Routers*.

functionality can be requested by giving the directory name terminated by a slash after the *save* command, for example:

```
save separated/messages/
```

There are several different possible formats for such deliveries; see the section "The appendfile Transport," in Chapter 9, for details. If this functionality is not enabled, the use of a pathname ending in a slash causes an error.

The pipe Command

```
pipe command
example: pipe "$home/bin/countmail $sender_address"
```

This command causes a separate process to be run, and a copy of the message is passed to it on its standard input. More than one *pipe* command may appear; each one causes a copy of the message to be written to its argument pipe, provided they are different (duplicate *pipe* commands are ignored).

The command supplied to *pipe* is split up by Exim into a command name and a number of arguments, delimited by whitespace, except for arguments enclosed in double quotes. In this case, backslash is interpreted as an escape, or if the arguement is enclosed in single quotes, no escaping is recognized. Note that as the whole command is normally supplied in double quotes, a second level of quoting is required for internal double quotes. For example:

```
pipe "$home/myscript \"size is $message_size\""
```

String expansion is performed on the separate components after the line has been split up, and the command is then run directly by Exim; it is not run under a shell. Therefore, substitution cannot change the number of arguments, nor can quotes, backslashes, or other shell metacharacters in variables cause confusion. Documentation for some programs that are normally run via this kind of pipe often suggest that the command should start with:

```
IFS=" "
```

This is a shell command, and should *not* be present in Exim filter files, since it does not normally run the command under a shell.

The preceding paragraph assumes a default configuration for the **pipe** transport that is being used. If `use_shell` is set on the transport, things are different. A number of other options on the transport can affect the way the command is run, including applying restrictions as to which commands may be run, and how any output from the command is handled. See the section "The pipe Transport," in Chapter 9 for details.

Ignoring Delivery Errors

As explained earlier in this chapter, filtering just sets up addresses for delivery; no deliveries are actually done while a filter file is active. If any of the generated addresses subsequently suffers a delivery failure, an error message is generated in the normal way. However, if the filter command that sets up a delivery is preceded by the word `noerror`, locally detected errors for that delivery, and any deliveries consequent on it (that is, from alias, forwarding, or filter files it invokes) are ignored. For example, suppose a user wants to scan all incoming messages using some program, as well as having them delivered into the normal mailbox. A command such as:

```
unseen noerror pipe $home/bin/mailscan
```

can be used; `unseen` ensures that normal delivery is not affected, and `noerror` ensures that a failure of the pipe does not cause a bounce message to generate.

Mail Commands

There are two commands that cause the creation of a new mail message, neither of which are significant actions unless the command is preceded by the word `seen`. Sending messages automatically is a powerful facility, but it should be used with care, because of the danger of creating infinite sequences of messages. The system administrator can forbid the use of these commands in users' filter files, by setting `forbid_filter_reply` on the **forwardfile** director.

To help prevent runaway message sequences, these commands have no effect when the incoming message is a bounce message, and messages sent by this means are treated as if they were reporting delivery errors (that is, their envelope senders are empty). Thus, they should never themselves cause a bounce message to be returned. The basic mail-sending command is:

```
mail to address-list
     cc address-list
     bcc address-list
     from address
     reply_to address
     subject text
     text text
     [expand] file filename
     return message
     log log file name
     once note file name
     once_repeat time interval
```

For example:

```
mail text "Got your message about @$h@_subject:"
```

All of the keywords that can follow *mail* are optional; you need only specify those whose defaults you want to change. As a convenience for use in one common case, there is also a command called *vacation*. It behaves in the same way as *mail*, except that the defaults for the `file`, `log`, `once`, and `once_repeat` keywords are:

```
expand file .vacation.msg
log  .vacation.log
once .vacation
once_repeat 7d
```

respectively. These are the same filenames and repeat period used by the traditional Unix *vacation* command. The defaults can be overridden by explicit settings, for example:

```
vacation once_repeat 14d
```

The *vacation* command is normally used conditionally, subject to the `personal` condition so as not to send automatic replies to nonpersonal messages from mailing lists or elsewhere.

For both commands, the key/value argument pairs can appear in any order. At least one of `text` or `file` must appear (except with *vacation*, where `file` can be defaulted); if both are present, the text string appears first in the message. If `expand` precedes `file`, each line of the file is subject to string expansion as it is included in the message.

If no `to` keyword appears, the message is sent to the address in the $reply_address variable, which is the contents of the *Reply-To:* header line if it exists, or otherwise the contents of the *From:* header line. An *In-Reply-To:* header is automatically included in the created message, giving a reference to the message identification of the incoming message.

Several lines of text can be supplied to `text` by including the escape sequence `\n` in the string where newlines are required, for example:

```
mail text
  "This is an automatic response to your \
  message.\nI am very busy, but will look \
  at it eventually."
```

If the command is output during filter file testing, newlines in the text are shown as `\n`.

Note that the keyword for creating a *Reply-To:* header line is `reply_to`, because Exim keywords may contain underscores, but not hyphens. If the `from` keyword is present and the given address does not match the user under which the filter is running, Exim adds a *Sender:* header line to the message.* If `from` is not specified, a *From:* header is constructed from the login name and the value of `qualify_domain`.

If `return_message` is specified, the incoming message that caused the filter file to be run is added to the end of the newly created message, subject to the maximum size limitation for returned messages, which is controlled by the `return_size_limit` option in Exim's main configuration.

If a log file is specified, an entry is added to it for each message sent. The entry contains several lines as in this example:

```
2000-05-01 14:04:06
To: John Doe <jd@somewhere.example>
Subject: Re: wish list for exim
```

If a `once` file is specified, it is used to hold a database for remembering who has received a message, and no more than one message is ever sent to any particular address, unless `once_repeat` is set. This specifies a time interval after which another copy of the message may be sent. For example:

```
once_repeat = 5d4h
```

causes a new message to be sent if five days and four hours have elapsed since the last one was sent. There must be no whitespace in a time interval.

The filename specified for `once` is used as the basename for direct-access (DBM) file operations. Exim creates and maintains the DBM file or files automatically. There are a number of different DBM libraries in existence. Some operating systems provide one as a default, but even in this case a different one may have been used when building Exim. With some DBM libraries, specifying `once` results in two files being created, with the suffixes *.dir* and *.pag* being added to the given name. With some others a single file with the suffix *.db* is used, or the name is used unchanged.

If `once` is used in a "vacation" scenario, the DBM file must be deleted by the user when the vacation is over and the filter file has been changed so as not to send any more messages.

More than one *mail* or *vacation* command may be obeyed in a single filter run; they are all honored, even when they are to the same recipient.

* See the `local_from_check` and `local_from_prefix` options in the section "Sender Address," in Chapter 13, *Message Reception and Policy Controls*, for ways to change this behavior.

Logging Commands

A log can be kept of actions taken by a filter file. For user filters, the system administrator may choose to disable this feature by setting `forbid_log` on the **forwardfile** director. Logging takes place while the filter file is being interpreted. It does not queue up for later as the delivery commands do. The reason for this is so that a log file need be opened only once for several write operations.

There are two commands, neither of which constitutes a significant action. The first defines a file to which logging output is subsequently written:

```
logfile file name
example: logfile $home/filter.log
```

The filename may optionally be followed by a mode for the file, which is used if the file has to be created. For example:

```
logfile $home/filter.log 0644
```

The number is interpreted as octal, even if it does not begin with a zero. The default for the mode is `0600`. It is suggested that the *logfile* command should normally appear as the first command in a filter file. Once *logfile* has been obeyed, the *logwrite* command can be used to write to the log file:

```
logwrite "some text string"
example: logwrite "$tod_log $message_id processed"
```

It is possible to have more than one *logfile* command, to specify writing to different log files in different circumstances. Writing takes place at the end of the file, and a newline character is added to the end of each string if there is not one already. Newlines can be put in the middle of the string by using the `\n` escape sequence. Lines from simultaneous deliveries may get interleaved in the file, as there is no interlocking, so you should plan your logging with this in mind. However, data should not get lost.

The testprint Command

It is sometimes helpful to be able to print out the values of variables when testing filter files. The command:

```
testprint "text"
example: testprint "home=$home reply_address=$reply_address"
```

does nothing when mail is being delivered. However, when the filtering code is being tested by means of the `-bf` or `-bF` options, the value of the string is written to the standard output.

The finish Command

The command *finish*, which has no arguments, causes Exim to stop interpreting the filter file. What happens next depends on whether any significant actions have been taken. The *finish* command itself is not a significant action unless preceded by the word `seen`. Reaching the end of a filter file has the same effect as obeying a *finish* command.

Obeying Filter Commands Conditionally

Most of the power of filtering comes from the ability to test conditions and obey different commands depending on the outcome. The *if* command is used to specify conditional execution, and its general form is:

```
if    condition
then  commands
elif  condition
then  commands
else  commands
endif
```

There may be any number of *elif-then* sections (including none) and the *else* section is also optional. Any number of commands, including nested *if* commands, may appear in any of the *`commands`* sections.

Conditions can be combined by using the words `and` and `or`, and parentheses can be used to specify how several conditions are to combine. Without parentheses, `and` is more strongly binding than `or`. Here is an example that uses parentheses:

```
if $h_subject: contains [EXIM] and
  (
  $return_path is exim-users-admin@exim.org or
  $h_to:$h_cc: contains exim-users@exim.org
  )
then
  save $home/Mail/exim-list
endif
```

A condition can be preceded by `not` to negate it, and there are also some negative forms of condition that are more English-like. For example:

```
if not personal and $h_subject: does not contain [EXIM]
  then save $home/Mail/other-lists
endif
```

The following descriptions show just the individual conditions, not the complete *if* commands.

String Testing Conditions

There are a number of conditions that operate on text strings, using the words `begins`, `ends`, `is`, `contains`, and `matches`. If the condition names are written in lowercase, the testing of letters is done without regard to case; if they are written in uppercase (for example, `CONTAINS`), the case of letters is significant.

```
text1 begins text2
text1 does not begin text2
example: $header_from: begins "Friend@"
```

A `begins` test checks for the presence of the second string at the start of the first, with both strings having been expanded.

```
text1 ends text2
text1 does not end text2
example: $header_from: ends "public.example.com"
```

An `ends` test checks for the presence of the second string at the end of the first, with both strings having been expanded.

```
text1 is text2
text1 is not text2
example: $local_part_suffix is "-foo"
```

An `is` test does an exact match between the strings, having first expanded both of them.

```
text1 contains text2
text1 does not contain text2
example: $header_subject: contains "evolution"
```

A `contains` test does a partial string match, having expanded both strings.

```
text1 matches text2
text2 does not match text2
example: $sender_address matches "(Bill|John)@"
```

For a `matches` test, after expansion of both strings, the second one is interpreted as a regular expression, and matched against the first. Care must be taken if you need a backslash in a regular expression, because backslashes are interpreted as escape characters both by the string expansion code and by Exim's normal string reading code when a string is given in quotes. For example, if you want to test the sender address for a domain ending in *.com*, the regular expression is:

```
\.com$
```

The backslash and dollar sign in that expression have to be escaped when used in a filter command, because otherwise they would be interpreted by the expansion code. Thus, what you actually write is:

```
if $sender_address matches \\.com\$
```

However, if the expression is given in quotes (mandatory only if it contains whitespace) you have to write:

```
if $sender_address matches "\\\\.com\\$"
```

with `\\\\` for a backslash and `\\$` for a dollar sign. Hence, if you actually require the string `\$` in a regular expression that is given in double quotes, you need to write `\\\\\\$`.

If the regular expression contains parenthesized subexpressions that capture parts of the matching string, numeric variable substitutions such as $1 can be used in the subsequent actions after a successful match. If the match fails, the values of the numeric variables remain unchanged. Previous values are not restored after *endif*; in other words, only one set of values is ever available. If the condition contains several subconditions connected by `and` or `or`, it is the strings extracted from the last successful match that are available in subsequent actions. Numeric variables from any one subcondition are also available for use in subsequent subconditions, since string expansion of a condition occurs just before it is tested.

Numeric Testing Conditions

The following conditions are available for performing numerical tests:

```
number1 is above number2
number1 is not above number2
number1 is below number2
number1 is not below number2
example: $message_size is not above 10k
```

The *number* arguments must expand to strings of digits, optionally followed by one of the letters K or M (uppercase or lowercase), which cause multiplication by 1024 and 1024×1024, respectively.

Testing for Personal Mail

A common requirement in user filters is to distinguish between incoming personal mail and mail from a mailing list. In particular, this test is normally required before sending "vacation messages," so as to avoid sending them to mailing lists. The condition:

```
personal
```

is a shorthand for:

```
$header_to: contains $local_part@$domain and
$header_from: does not contain $local_part@$domain and
$header_from: does not contain server@ and
$header_from: does not contain daemon@ and
$header_from: does not contain root@ and
$header_subject: does not contain "circular" and
```

```
$header_precedence: does not contain "bulk" and
$header_precedence: does not contain "list" and
$header_precedence: does not contain "junk"
```

This condition tests for the appearance of the current user in the *To:* header, checks that the sender is not the current user or one of a number of common daemons, and checks the content of the *Subject:* and *Precedence:* headers. It is useful only in user filter files, because a unique recipient (from which to set $local_part and $domain) does not exist during the running of a system filter.

If prefixes or suffixes are in use for local parts (something that depends on the configuration of Exim), the first two tests shown previously are also done with:

```
$local_part_prefix$local_part$local_part_suffix
```

instead of just $local_part. If the system is configured to rewrite local parts of mail addresses (for example, to rewrite *dag46* as *Dirk.Gently*), the rewritten form of the address is also used in these tests.

This example shows the use of `personal` in a filter file that is sending out vacation messages:

```
if personal then
  mail
   to $reply_address
   subject "Re: $h_subject:"
   file $home/vacation/message
   once $home/vacation/once
   once_repeat 10d
endif
```

It is quite common for people who have mail accounts on a number of different systems to forward all their mail to one host, and in this case a check for personal mail should test all their mail addresses. To allow for this, the `personal` condition keyword can be followed by:

```
alias address
```

any number of times, for example:

```
if personal
  alias smith@else.where.example
  alias jones@other.place.example
then ...
```

This causes messages containing the alias addresses to be treated as personal. The aliases are used when checking both the *To:* and the *From:* headers.

Testing for Significant Actions

Whether or not any previously obeyed filter commands have resulted in a significant action can be tested by the condition `delivered`, for example:

```
if not delivered then save mail/anomalous endif
```

Testing for Error Messages

The condition `error_message` is true if the incoming message is a mail delivery error message (bounce message), that is, if its envelope sender address is empty. Putting the command:

```
if error_message then finish endif
```

at the head of a filter file is a useful insurance against things going wrong in such a way that you cannot receive delivery error reports, and it is highly recommended. Note that `error_message` is a condition, not an expansion variable, and therefore is not preceded by `$`.

Testing Delivery Status

There are two conditions that are intended mainly for use in system filter files but that are available in users' filter files as well. The condition `first_delivery` is true if this is the first attempt to deliver the message, and false otherwise.

The condition `manually_thawed` is true only if the message was frozen for some reason, and was subsequently released by the system administrator. It is unlikely to be of use in users' filter files. An explicit forced delivery counts as a manual thaw, but thawing as a result of the `auto_thaw` option does not.

Testing a List of Addresses

There is a facility for looping through a list of addresses and applying a condition to each of them. It is executed as a condition that shapes part of an *if* command, and takes the form:

```
foranyaddress string (condition)
```

where *`string`* is interpreted as a list of RFC 822 addresses, as in a typical header line or the value of $recipients in a system filter, and *`condition`* is any valid filter condition or combination of conditions. The parentheses surrounding the condition are mandatory to delimit it from possible further subconditions of the enclosing *if* command. Within the condition, the expansion variable $thisaddress is set to the noncomment portion of each of the addresses in the string in turn. For example, if the string was:

```
B.Simpson <bart@springfield.example>,lisa@springfield.example (his sister)
```

then $thisaddress would take on the values `bart@springfield.example` and `lisa@springfield.example` in turn.

If there are no valid addresses in the list, the whole condition is false. If the internal condition is true for any one address, the overall condition is true and the loop ends. If the internal condition is false for all addresses in the list, the overall condition is false. In other words, the overall condition succeeds if, and only if, at least one address in the list satisfies the internal condition.

This example tests for the presence of an eight-digit local part in any address in a *To:* header:

```
if foranyaddress $h_to: ( $thisaddress matches ^\\d{8}@ ) then ...
```

When the overall condition is true, the value of $thisaddress in the commands that follow `then` is the last value it took inside the loop. At the end of the *if* command, the value of $thisaddress is reset to what it was before. It is best to avoid the use of multiple occurrences of `foranyaddress`, nested or otherwise, in a single *if* command, if the value of $thisaddressis to be used afterwards, because it is not always clear what the value will be. Nested *if* commands should be used instead.

Header lines can be joined together if a check is to be applied to more than one of them. For example:

```
if foranyaddress $h_to:,$h_cc: ....
```

scans through the addresses in both the *To:* and the *Cc:* headers.

Additional Features for System Filters

During the running of a system filter, the variable $recipients contains a list of all the envelope recipients of the message, separated by commas and whitespace. In addition, for any deliveries set up by *deliver*, *save*, or *pipe* commands in a system filter, an extra header line is added to the message, with the name *X-Envelope-To:*. This contains up to 100 of the message's original envelope recipients.

There are also some additional filtering commands. These are normally permitted only in system filters, and attempts to use them in a user filter, or when testing using `-bf`, are faulted. However, there is an option for the **forwardfile** director called `allow_system_actions`, which allows the *fail* and *freeze* commands to be used in a nonsystem filter. This is intended for use on systems where centrally managed per-user filter files are run; it is not normally sensible to enable it when users can modify their own filter files.

The fail Command

```
fail text "text"
example: fail text "Administrative rejection"
```

This command prevents any deliveries of the message from taking place, except for those that may have previously been set up in the filter. In this way it is similar to `seen finish`, but in this case a bounce message is generated that contains the text string (which can be omitted if not required). No further commands in the filter file are obeyed, so if, for example, you want to use *logwrite* to keep a log of forced failures, you must place that command before *fail*.

Take great care with the *fail* command when basing the decision to fail on the contents of the message, because the normal delivery error bounce message includes the contents of the original message, and will therefore trigger the *fail* command again (causing a mail loop) unless steps are taken to prevent this. Testing the `error_message` condition is one way to prevent this. You could use, for example:

```
if $message_body contains "this is spam"
  and not error_message
then
  fail text "spam is not wanted here"
endif
```

The alternative is clever checking of the body or header lines to detect error messages caused by the filter.

The freeze Command

```
freeze text "text"
example: freeze text "Administrative rejection"
```

This command also prevents any deliveries other than those previously set up in the filter from taking place, but the message is not bounced. Instead, after any deliveries that were set up by the filter have been attempted, the message remains in the queue and is frozen. No further commands in the filter file are obeyed, so if, for example, you want to use *logwrite* to keep a log of freezing actions, you must place that command before *freeze*.

The *freeze* command is ignored if the message has been manually unfrozen and not subsequently manually frozen. This means that automatic freezing in a system filter can be used as a way of checking out suspicious messages. If a frozen message is found on inspection to be valid after all, manually unfreezing it (using the `-Mt` option, or via the Exim monitor) allows it to be delivered. A forced delivery attempt (using `-M` or `-qff`, for example) also counts as manual unfreezing, but reaching the `auto_thaw` time does not.

The headers add Command

```
headers add "text"
example: headers add "X-Filtered: checked on $primary_hostname"
```

The string is expanded and added to the end of the message's header lines. It is the responsibility of the filter maintainer to make sure it conforms to RFC 822 syntax. Leading whitespace is ignored, and if the string is otherwise empty, or if the expansion is forced to fail, the command has no effect. A newline is added at the end of the string if it lacks one. More than one header line may be added in one command by including `\n` within the string.

These header lines that are added by a system filter are visible during the subsequent delivery process, and can be referred to in expansion strings. This is different from header lines that are added by drivers, which apply only to individual addresses, and are not generally visible.

The headers remove Command

```
headers remove "text"
example: headers remove "Return-Receipt-To"
```

The string is expanded, and is then treated as a colon-separated list of header names. Any header lines with those names are removed from the message. This command applies only to the original header lines that are stored with the message; others such as *Envelope-To:* and *Return-Path:* that are added at delivery time cannot be removed by this means.

The header lines that are removed by a system filter become invisible during the subsequent delivery process, and cannot be referred to in expansion strings. This is different from headers that are specified for removal by drivers, which remain visible and are removed only when the message is transported.

11

Shared Data and Exim Processes

In the overview of Exim's operation in Chapter 3, *Exim Overview*, the use of multiple processes is mentioned. In this chapter, we are going to describe the different types of process that are used for handling messages. This is background information about the way Exim works, which may be useful when you want to understand exactly what it is doing. The four process types are as follows:

The daemon process
: A daemon process listens for incoming SMTP connections, and starts a reception process for each one. The daemon may also periodically start queue runner processes. There is only one daemon process.

Reception processes
: A reception process accepts an incoming message and stores it on Exim's spool disk.

Queue runner processes
: A queue runner process scans the list of waiting messages, and starts a delivery process for each one.

Delivery processes
: A delivery process performs one delivery attempt on a single message.

Most Exim processes are short-lived. They perform one task, such as receiving or delivering one message, and then exit. The only exception is the daemon process, which, as its name implies, runs continuously. Each Exim process that is receiving or delivering a message operates, for the most part, independently of any other Exim process. Nevertheless, Exim processes do interact with each other by referring to shared files. We describe these first, because their contents affect the way the processes operate. The files fall into three categories, as shown in Figure 11-1.

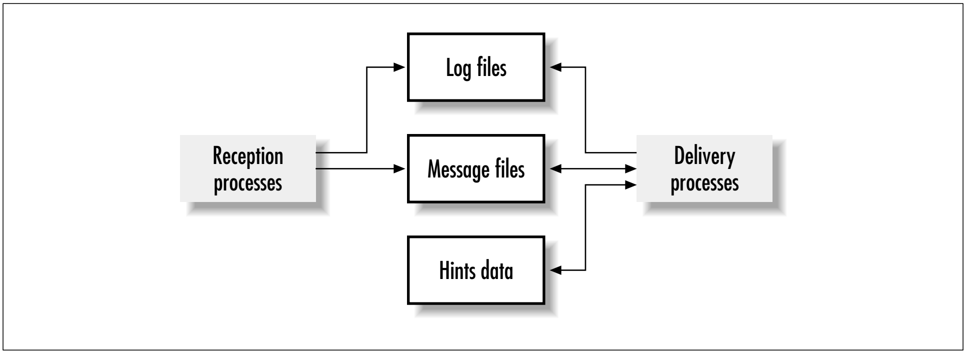


Figure 11-1. Shared data

Log files
Record actions that Exim has taken.

Message files
Contain the messages that are in the process of being received or delivered.

Hints files
Contain information about delivery problems that were encountered earlier.

The message files and hints files are held in subdirectories inside Exim's spool directory, whose name is configurable, though it is most commonly called */var/spool/exim*. You can find out the name of the spool directory by running the command:

```
exim -bP spool_directory
```

In some configurations, the log files are here as well, in another subdirectory, but Exim can be configured to put them elsewhere. In configurations where the log information is being written only to *syslog*, there are no log files.

Message Files

Each message is held in two separate files, whose names consist of the message's unique identifier followed by `-D` and `-H`, respectively. For example, the message `12b2ie-0000GI-00` is held in the two files:

```
12b2ie-0000GI-00-D
12b2ie-0000GI-00-H
```

The first line of each file is the file's own name. This self-identification is an insurance against a disk crash that destroys the directory but not the files themselves. The *-D* file contains the body of the message and may be very large. It is never updated after the message has been received.

The *-H* file contains the message's envelope and header lines, together with other information about the message, such as the host it arrived from and those addresses to which it has already been delivered. The exact format is described in the reference manual. This file is normally fairly small, and is updated during the lifetime of a message that cannot be delivered to all its recipients at the first attempt, in order to record which addresses have been dealt with. The existence of an *-H* file is sufficient to indicate the presence of a message on the queue. No separate list of messages is maintained.

The *input* directory within Exim's spool directory is used to hold the message files, which under normal circumstances are continually being created and deleted as mail passes through the system. If a large amount of mail is being handled, there will be contention between different Exim processes for access to the *input* directory. Furthermore, if the total number of messages on the queue at any time is large, updating the directory may take a long time, and thus may lead to poor performance. The size of the directory that provokes a degradation in performance varies between different operating systems. Solaris, for example, can handle bigger directories than Linux (using its default file system) before it starts to degrade. To improve matters when the queue is large, the option:

```
split_spool_directory = true
```

can be set. When this is done, an extra directory level is used. Within the *input* directory, 62 subdirectories are created, with names consisting of a single letter or digit, and the message files are distributed among them. The sixth character in the message ID is used to determine the subdirectory in which a message is stored. For example, the message `12b2ie-0000GI-00` is stored in *input/e*. The sixth character is chosen because it is the least significant base-62 digit of the time of the message's arrival and changes every second. Splitting the *input* directory like this reduces the number of files in any one directory, and also reduces the amount of contention between processes that are trying to create or delete files in the directory. One disadvantage is that more work is required to scan the queue, either for listing it or for performing a queue run, but these operations are relatively infrequent.

During delivery, Exim creates two other files for each message. A third file in the *input* directory (or a subdirectory thereof) with suffix *-J* (for "journal") is used to record each recipient address as soon as it is delivered. If the message has to be retained for a subsequent delivery attempt at the end of a delivery run, then the contents of the *-J* file are merged into the *-H* file, and the *-J* file is deleted. There are two reasons for the use of journal files:

- If there is a crash of any sort between the time a delivery completes and the time that this fact is recorded on disk, an unwanted second copy of the message will be delivered later.* Appending a single line to the *-J* file is a quick operation compared with updating the *-H* file, which involves writing a number of pieces of data to a new file and then renaming it. The use of a journal therefore minimizes the possibility of duplicate deliveries.
- Another reason for using a journal is that when several remote deliveries of the same message are taking place in parallel, and in multiple processes, they can all add to the same *-J* file easily, without the need for explicit locking, because the operating system ensures that only one update happens at once.

The final per-message file is the *message log*, which contains log entries that record the progress of the message's delivery. For example, for every successful or unsuccessful delivery, a line is written to this file. The same information is also recorded on Exim's main log, but keeping copies in message log files makes it easy for an administrator to check the progress, or lack of it, of individual messages. Message logs are kept in a subdirectory called *msglog* in Exim's spool directory, and their names are the relevant message IDs. A message's log file is deleted when processing of the message is complete.† If the `split_spool_directory` option is set, the *msglog* directory is subdivided in the same way as the *input* directory.

Locking Message Files

When an Exim process is working on a message, it locks the *-D* file to prevent any other Exim process from trying to deliver the same message.‡ If another delivery process is started for the message, for example, by a queue runner, it notices the lock, and exits without doing anything, logging the message:

```
Spool file is locked
```

This is a normal occurrence and does not indicate an error, though if it continues to occur for the same message over a long period of time, it might mean that there is some problem with the delivery process. You can turn off these log entries by setting the log level less than 5.

* Even if Exim and all the other software in the system were bug-free, hardware failures and power losses can cause this effect.

† There is an option called `preserve_message_logs`, which causes them to be moved to a spool directory called *msglog.OLD* instead for statistical or debugging uses. It is then the administrator's responsibility to ensure that they get deleted.

‡ The *-H* file cannot be used for locking a message, because it can be updated during a delivery process, and the updating is carried out by writing a temporary file and renaming. This changes the underlying identity (inode) of the file, which is what locking is based on.

Hints Files

Exim collects data about previously encountered problems, in order to adapt its behavior to changing circumstances. It remembers, for example, the hosts to which it has been unable to connect, so as not to keep trying them too often. The term "hints" is used to describe this data, because it is not critical for Exim's operation. If, for example, information about a failing host is lost, Exim will try to deliver to it at the next opportunity, instead of waiting for a previously calculated retry time, but this just means that the system is doing more work than it would otherwise have done. The pattern of delivery attempts is affected, but no messages are damaged or lost.

Because of the noncritical nature of this data, Exim maintains it by simple system I/O calls; there is no need for the sort of heavyweight transaction-based apparatus that would be necessary if the data had to be managed in the safest possible manner. This means that the overhead of maintaining the hints is minimal.

The hints data is kept in a number of files in a subdirectory of the spool directory called *db*. These files are not read or written sequentially like conventional files, because that would be very slow. Instead, the data they contain is held in indexed DBM files. Exim uses four different kinds of hints database, as follows:

- The *retry* database holds information about temporary failures, which can be related to a particular host, mail address, or message (sometimes to a combination of host and message). The database records the type of error, the time of the first failure, the time of the last delivery attempt, and the earliest time it is reasonable to try again. A discussion of how this data is created and used is given in Chapter 12, *Delivery Errors and Retrying*.
- There is a database that contains lists of messages that are awaiting delivery to specific hosts, after having failed at their first attempt. In normal circumstances, this is updated only when a message fails to be delivered, though it is possible to force new messages to be added before their first delivery attempt.* The data is keyed by hostname and IP address, and consists of a list of message IDs. The name of the database is *wait-*, followed by the name of the transport that attempted the delivery. A discussion of how this data is created and used is also given in Chapter 12.
- The *reject* database is used to remember certain types of message rejection so that if the same host tries to send you the same message again, alternative means of rejection can be tried. This gets round the problem of some SMTP clients that contravene the RFCs by retrying after certain kinds of permanent

* See the `queue_smtp_domains` and *-odqs* options.

rejection. A discussion of the details can be found in the section "Temporary Sender Verification Failures," in Chapter 13, *Message Reception and Policy Controls*.

- The final database is used in conjunction with the SMTP `ETRN` command. This command allows a connected client host to request the server to attempt to deliver mail for a specific domain using a special kind of queue runner process. `ETRN` is mainly of use when the client is a dial-up host. The database is used to ensure that a client host cannot cause more than one queue runner process to run at once by issuing multiple `ETRN`s. Its name is *serialize-etrn* and its use is discussed in the section "The ETRN Command," in Chapter 15, *Authentication, Encryption, and Other SMTP Processing*.

The exact names of the files in the *db* directory depend on the DBM library that is in use. Two filenames are used by some libraries, with the extensions *.dir* and *.pag*, whereas others use *.db* or no extension at all. Thus, the *retry* database, for example, might be held in *retry.dir* and *retry.pag*, or *retry.db*, or just *retry*. You will also see files whose names end in *.lock* in the *db* directory. These are used by Exim when updating the hints files to ensure that only one Exim process writes to a database at once.*

Every delivery process consults the *retry* hints, and after any SMTP deliveries, the *wait-* hints are checked. If the system is running normally, the majority of these accesses are read-only, using shared locks that do not hold processes up. Only when there is a temporary failure or a successful delivery after a previous failure, is it necessary for a process to gain exclusive access in order to update the hints. The hope is that this is a relatively rare occurrence.

Hints files accumulate out-of-date information that needs to be cleared out from time to time. For example, if a number of hosts are set up to accept mail for a certain domain, but all are unreachable at some time, retry data for each host is created. However, when the waiting message is subsequently delivered to one of them, the information for the others remains. There is a utility called *exim_tidydb* that clears away "dead wood" (data that has not been updated for a long time) in hints files; it should be run at regular intervals (for instance, daily or weekly). There are also some other utilities for inspecting and modifying the contents of hints files; these are described in the section "Hints Database Maintenance," in Chapter 21, *Administering Exim*.

* Conventional locking of the files themselves cannot be used because Exim accesses them indirectly via a DBM library, and not all the DBM libraries provide integrated locking facilities. For simplicity, therefore, an external lock is used.

Log Files

Unless Exim has been configured to use only *syslog*, it writes logging data to three files in its log directory, whose location is configurable at build time or at runtime. The most common locations are the *log* subdirectory within Exim's spool directory (which is the default), or a location such as */var/log/exim* on systems that keep all their logs in one place. The contents of Exim's logs are described in the section "Log Files," in Chapter 21.

Exim processes all write to the same log files, but no Exim process ever reads any log data. Interlocking between processes is achieved by opening log files for appending, and ensuring that each log line is written in a single write operation. The operating system then ensures that only one update happens at once, and there is no need for the processes to do any locking of their own.

Log files are normally cycled on a regular (usually daily) basis by renaming. An Exim utility called *exicyclog* (see the section "Cycling Log Files," in Chapter 21) is provided for this purpose. Afterwards, a new file is created as soon as any Exim process has something to be logged. Existing Exim processes that have already opened the old file may keep it open, but will not write to it any more; as soon as these processes have completed, the file becomes dormant. A common practice is to compress the previous-but-one log file 24 hours after it was renamed; the *exicyclog* script does this automatically.

User and Group IDs for Exim Processes

Exim is normally installed as a "setuid root" program, with permissions set like this:

```
    -rwsr-xr-x   1 root   mail    561952 Nov 30 09:53 exim
```

The `s` permission means that whenever Exim starts up, it acquires *root* privilege, without which, for example, it cannot write into every user's mailbox. However, it is desirable on general security principles that any Exim process should stop running as *root* as soon as it no longer needs the privilege. In order to do this, it needs to have some other uid to use instead.

When Exim is built, a uid and gid can be defined for it to use when it no longer needs any privilege. These are referred to in this book as "the Exim uid" and "the Exim gid." A general discussion of security can be found in the section "Security Issues," in Chapter 19, *Miscellany*.

Process Relationships

Although there is no central process that has overall control of what Exim is doing, Exim processes do interact with each other in various ways. The connections are illustrated in Figure 11-2. The upper part of the figure is concerned with message reception, and the lower part with delivery. The solid lines indicate data flows, while dashed ones are used to show where one process creates another without passing any message data. For example, the daemon process creates a new process for each incoming SMTP call, and local reception processes can be created by any other process such as a user's MUA or a script that sends a message.

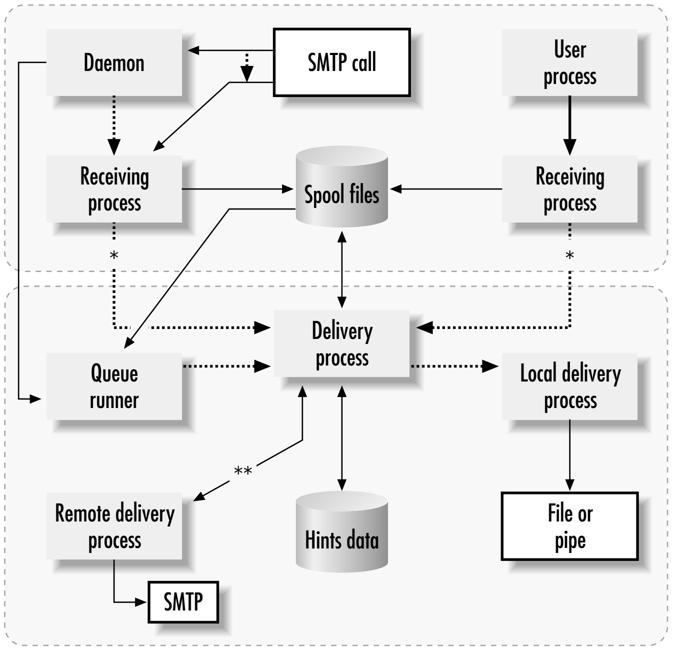


Figure 11-2. Process relationships

Delivery processes are started as a result of the arrival of a new message (the two lines marked *), or by a queue runner process. Exim has command-line options that can be used by privileged users or by some automatic mechanism external to Exim itself to start delivery processes or queue runners. Immediate delivery can be suppressed by setting options in the configuration file, either unconditionally or under specific circumstances (see the section "Reception Processes," later in this chapter).

The forking of separate processes for remote deliveries, as indicated by the line marked **, happens only if parallel remote delivery is configured and there is more than one remote host to which copies of the message are to be sent. Otherwise, remote delivery is done in the main delivery process, and if several remote hosts are involved, they are contacted one at a time.

The Daemon Process

The daemon performs two tasks: listening for incoming SMTP calls and periodically starting queue runner processes. It is possible to run two separate daemons for these tasks, but there seems little advantage in doing so. This description assumes the common style of usage, where the daemon is started by a command such as:

```
exim -bd -q15m
```

The `-bd` option sets up a daemon that is listening for incoming SMTP, while `-q15m` specifies that it should start a new queue runner process every 15 minutes. These options are compatible with Sendmail, so the command:

```
/usr/sbin/sendmail -bd -q15m
```

that appears in the boot scripts in most operating systems will correctly start an Exim daemon, provided that */usr/sbin/sendmail* has been converted into a symbolic link to the Exim binary.

When a daemon starts up, unless debugging is enabled, it disconnects itself from any controlling terminal. After this, it writes its process ID into a file so that it can easily be found. The location of this file is configured either at build time, or by setting `pid_file_path`, but if no location is specified, the pid file is written in Exim's spool directory using the name *exim-daemon.pid*. This makes it easy to find when you want to kill off the daemon, or send it a HUP signal after changing Exim's configuration file.

If you are running a host that uses different IP addresses (on virtual interfaces) to support a number of virtual web servers, you may not want to have incoming SMTP connections on all the virtual interfaces. If you set `local_interfaces` to a list of IP addresses, the daemon listens only on those interfaces. Otherwise, all interfaces are used.

The same port number is used in all cases. It defaults to the standard SMTP port (port 25), but can be changed by the `daemon_smtp_port` option or by the use of *-oX* on the command line that starts the daemon. Other ports can be useful for special applications, or for testing.

The following checks are performed when the daemon receives an incoming SMTP call:

- If the maximum permitted number of simultaneous incoming SMTP calls, as set by the `smtp_accept_max` option, is exceeded, the call is rejected with the error response:

  ```
  421 Too many concurrent SMTP connections; please try again later.
  ```

  You should set `smtp_accept_max` to a value that is appropriate to the power of your host and the speed of your network connection. The default value of 20 is on the small side.

- If `smtp_accept_max_per_host` is set, the list of current connections is scanned to find out how many are from the incoming IP address. If the new connection would cause the limit to be exceeded, the call is rejected with the error:

  ```
  421 Too many concurrent SMTP connections from one
      IP address; please try again later.
  ```

  There is no default limit on the number of connections from a single host.

- If `smtp_accept_queue` is greater than zero, and the number of incoming SMTP connections exceeds its value, no delivery processes are started for messages that are received. Instead, they remain on the queue until picked up by a queue runner process. This value is useful only if it is less than `smtp_accept_max`. It can help to keep the system load down at times of high SMTP input.

- If `smtp_accept_reserve` is greater than zero, Exim reserves that many incoming SMTP connections, from the maximum set in `smtp_accept_max`, for those hosts listed in `smtp_reserve_hosts`. For example, with the following:

  ```
  smtp_accept_max = 100
  smtp_accept_reserve = 15
  ```

  once there are 85 incoming connections, new ones are accepted only from the favored hosts. This can be useful on clusters of hosts that are interchanging mail (for example, the hosts on a LAN and an Internet gateway), because it prevents external hosts from using all the gateway's SMTP slots, and thereby blocks mail that comes from internal hosts.

- The value of `smtp_load_reserve` is a system load average.* When the system's actual one-minute load average is above this value, connections are accepted

* See the Unix *uptime* command for details of system load averages.

only from hosts that are defined by `smtp_reserve_hosts` (if any). Other hosts are rejected with the error:

```
421 Too much load; please try again later.
```

If none of the checks fail, a new process is created to continue handling the incoming call. The daemon is now ready to accept another SMTP call. Although it does very little processing before forking, other incoming calls may arrive during the time it is handling a call. The operating system maintains a queue of waiting calls, the length of which is specified by `smtp_connect_backlog`. Once this number of connections is waiting, subsequent connection attempts are supposed to be refused at the TCP/IP level. However, on some operating systems, attempts to connect have been seen to time out in such circumstances. The default value of 5 is a conservative one, suitable for older and smaller systems. For large systems, it is probably a good idea to increase this, possibly substantially (50 is a reasonable number).

Apart from incoming TCP/IP calls, there are two other events that wake up a daemon process. The first is its timer, in the case when it is configured to start queue runner processes periodically. Whenever the timer expires, a new queue runner process is created, unless `queue_run_max` is greater than zero and the number of queue runner processes that are still running is greater than or equal to its value.

The other useful event is the arrival of a SIGHUP signal. You should send a SIGHUP signal to the daemon whenever Exim's configuration file has been updated. The daemon reacts by closing down any sockets that it is listening on, and reexecuting itself, thereby rereading the configuration file. It is a good idea to check that the daemon has restarted successfully, by checking the end of the main log. A consequence of the way SIGHUP is handled is that the memory of the number of incoming SMTP calls and running queue runner processes is forgotten.

The daemon has to take notice of the completion of SMTP reception processes and queue runner processes in order to maintain its counts of active processes, so that it can refrain from starting new ones when the limits are reached. However, termination of these processes does not wake up the daemon. Instead, the daemon checks for completed processes whenever it wakes up for any other reason, which on busy systems happens frequently. On systems where not much is happening, "zombie" (defunct) processes that are children of the daemon can sometimes be seen. This is perfectly normal; they get tidied away the next time the daemon wakes up.

Summary of Options for the Daemon

Here is a summary of the configuration options that are relevant to the daemon process and the reception processes it creates:

`local_interfaces` (string, default = unset)
: The string must be a colon-separated list of IP addresses. If `local_interfaces` is unset, the daemon issues a generic `listen()` that accepts incoming SMTP calls on any interface. Otherwise, it listens only on the interfaces identified here. An error occurs if it is unable to bind a listening socket to any listed interface. Some systems set up large numbers of virtual interfaces in order to provide many different virtual web servers. In these cases, `local_interfaces` can be used to restrict incoming SMTP traffic to specific interfaces only. The contents of `local_interfaces` are also used as a list of the local host's addresses when routing mail and checking for mail loops (see the section "Domains That Route to the Local Host," in Chapter 8, *The Routers*).

`queue_run_max` (integer, default = 5)
: This controls the maximum number of queue-running processes that an Exim daemon will run simultaneously. This does not mean that it starts them all at once, but rather that if the maximum number are still running when the time comes to start another one, it refrains from starting it. This can happen with very large queues and/or very sluggish deliveries. Remember that Exim is not a centralized system. This limit applies only to a single daemon process. If a queue runner is started by another means (for example, by hand by the administrator), it does not count towards this limit. Also, if the daemon is restarted, it loses its memory of previously started queue runners.

`smtp_accept_max` (integer, default = 20)
: This specifies the maximum number of simultaneous incoming SMTP calls that Exim will accept. It applies only to the listening daemon; there is no control (in Exim) when incoming SMTP is being handled by *inetd*. If the value is set to zero, then no limit is applied. However, it must be nonzero if `smtp_accept_max_per_host` or `smtp_accept_queue` is set.

`smtp_accept_max_per_host` (integer, default = 0)
: This option restricts the number of simultaneous IP connections from a single host (strictly, from a single IP address) to the Exim daemon. Once the limit is reached, additional connection attempts are rejected with error code 421. The default value of zero imposes no limit. If this option is not zero, `smtp_accept_max` must also be nonzero.

`smtp_accept_queue` (integer, default = 0)
: If the number of simultaneous incoming SMTP calls handled via the listening daemon exceeds this value, messages received are simply placed on the queue, and no delivery processes are started automatically. A value of zero

implies no limit, and clearly any nonzero value is useful only if it is less than the `smtp_accept_max` value (unless that is zero). See also `queue_only`, `queue_only_load`, `queue_smtp_domains`, and the various *-od* command-line options.

`smtp_accept_reserve` (integer, default = 0)
When `smtp_accept_max` is set greater than zero, this option specifies a number of SMTP connections that are reserved for connections from the hosts that are specified in `smtp_reserve_hosts`.

`smtp_connect_backlog` (integer, default = 5)
This specifies a maximum number of waiting SMTP connections. Exim passes this value to the TCP/IP system when it sets up its listener. Once this number of connections are waiting for the daemon's attention, subsequent connection attempts are refused at the TCP/IP level.

`smtp_load_reserve` (fixed point, default = unset)
If the system load average ever gets higher than this, incoming SMTP calls are accepted only from those hosts that match an entry in `smtp_reserve_hosts`.

`smtp_reserve_hosts` (host list, default = unset)
This option defines hosts for which SMTP connections are reserved; see `smtp_accept_reserve` and `smtp_load_reserve`.

Reception Processes

Reception processes for handling incoming SMTP messages from remote hosts are started by the daemon, or by *inetd*.* A TCP/IP connection from a local process is treated in the same way as a connection from a remote host, even if it uses the loopback address 127.0.0.1 (or ::1 on an IPv6 system).

The other way a local process can send a message is by starting an Exim reception process itself (that is, by running */usr/sbin/sendmail* or */usr/lib/sendmail* in a new process), and passing the message on its standard input. There are several different ways in which this can be done; for details see the section "Messages from Local Processes," in Chapter 13.

Receiving a message consists of copying its contents to a pair of *-D* and *-H* files in Exim's spool area. Once these files have been successfully written, reception is complete, and Exim returns a success response to the sender.

The existence of an *-H* file signifies the presence of a message on the queue. There is no separately maintained list of messages; the files *are* the queue. As

* Most Exim installations use a daemon, and that is generally assumed in this book. Nevertheless, if an administrator wants to route all incoming TCP/IP connections through *inetd* for whatever reason, Exim can handle this way of working. However, it is likely to be less efficient on a busy system.

soon as the *-H* file comes into existence (by renaming from a temporary name), it is legitimate for another Exim process to start work delivering the message. A process that receives a message creates a delivery process before it finishes, except in the following circumstances:

- If `queue_only` is set, or the reception process was started with the *-odq* option, incoming messages are placed on the queue without an automatic delivery process being started. Deliveries then occur only via queue runner processes, or manual intervention. On very busy systems, this may lead to better throughput because the total number of delivery processes can be controlled.
- If `queue_only_load` is set to some positive value, and the system load is greater than this value, incoming messages are queued without immediate delivery. For example:

  ```
  queue_only_load = 8
  ```

  specifies that no immediate deliveries are to take place when the system load is greater than 8.
- If `queue_only_file` names an existing file, no immediate delivery takes place. This facility is intended for use on dial-up hosts, where a configuration such as the following:

  ```
  queue_only_file = /etc/not-dialed-up
  ```

  can be used in conjunction with commands in the dial-up starting and stopping scripts that remove and create the file. A colon-separated list of files may be given, in which case the existence of any one of them is enough to provoke queuing.

There are two other options which may delay some of the deliveries until the next queue run, while not delaying all of them. They are `queue_remote_domains` (or *-odqr*) and `queue_smtp_domains` (or *-odqs*); a full description is included in the section "Intermittently Connected Hosts," in Chapter 12. You can also make these kinds of queueing dependent on the existence of a specific file, by using one of the words `remote` or `smtp` before the filename for the `queue_only_file` option. For example:

```
queue_only_file = remote/etc/not-dialled-up
```

prevents immediate delivery of remote addresses when the file exists, while allowing local deliveries to proceed.

Queue Runner Processes

The job of a queue runner process is to start delivery processes for all the messages that are on Exim's spool, waiting for each one to complete before starting the next. In other words, a single queue runner process works its way through the queue, attempting to deliver just one message at time. It does not distinguish between messages that have suffered a failed delivery attempt and those that were put on the queue without an immediate delivery process being started. However, it does skip over frozen messages.

Notice that the queue runner does not itself do any of the work of delivering. That is left to the delivery processes that it creates. All the queue runner does is arrange for delivery processes to be started for all the waiting messages.

Queue runner processes should be started at regular intervals, and the most common way of doing this is to use an option such as *-q15m* (every 15 minutes) on the command that starts the Exim daemon. Multiple queue runner processes may be active simultaneously, and using a daemon it is possible to control the maximum number using the `queue_run_max` option. However, any suitable external mechanism (such as *cron*) can be used to start a single queue runner by means of the command:

```
    exim -q
```

In very busy environments it may be desirable to control the total number of Exim delivery processes that run simultaneously. If you set `queue_only`, immediate delivery on reception is suppressed, so the only delivery processes are those that are started by a queue runner. If you configure the daemon to start queue runners frequently (say every minute by starting it with *-q1m*), but limit the maximum number that may be simultaneously active, for example, by setting:

```
    queue_run_max = 10
```

there will only ever be 10 delivery processes running at once, though subprocesses may be created for local deliveries. Also, if you have set `remote_max_parallel`* greater than one, there may also be multiple subprocesses for remote deliveries.

Special Kinds of Queue Run

Additional queue running options are available for special purposes; these are not normally used in regular periodic queue runs, but are either specified for one-off queue runs by the system administrator, or generated in response to some specific event such as the connection of a dial-up host.

* See the section "Parallel Remote Delivery," in Chapter 4, *Exim Operations Overview.*

In a normal queue run, the delivery processes inspect the retry data for addresses and hosts, and refrain from attempting deliveries for those addresses whose retry times have not yet arrived. This can be overridden by starting a queue runner with *-qf*, which forces a delivery attempt for all addresses. An even more powerful option is *-qff*, which, in addition to overriding retry times, causes frozen messages not to be skipped. If `l` (the letter "L") is added to the end of any of the *-q* options, only local addresses (those that match `local_domains`) are considered for delivery. These variants of the *-q* option are summarized in Table 11-1.

Table 11-1. Queue Runner Options (-q)

Option	Meaning
-q	Single queue run, respect retry times
-qf	Single queue run, override retry times
-qff	Single queue run, override retry times, include frozen messages
-ql, *-qfl*, *-qffl*	Same as previous, but deliver only local domains

It is possible to run only part of the queue, or to select only messages whose senders or recipients match certain patterns. Details of the options to do this are given in Chapter 20, *Command-Line Interface to Exim*.

Normal queue runners process the waiting messages in an arbitrary order that is likely to be different each time. This is beneficial when there is one particular message that is provoking delivery failures for some address. For example, very large messages sometimes cause timeouts or other problems when transmitted to remote hosts, while smaller messages to the same hosts might get through. After a temporary error such as a timeout, Exim is not prepared to try the same host again for some time, so any other messages for the same address are likely to be passed over in the same queue run. By processing the messages "randomly," there is a chance that in some future queue run the trouble-free messages will be handled first, and so be delivered instead of being delayed behind the problem message.

You can specify that Exim should not do this randomizing, and instead should process the messages in order of their IDs, which is in effect the order of their arrival, by setting `queue_run_in_order`. However, there is rarely a good reason for doing this, and it can degrade performance on systems with a large queue where `split_spool_directory` is set. The reason is that all the subdirectories of the *input* directory have to be scanned before any deliveries can start, in order to obtain the complete list of messages and sort them by ID. When the order is not constrained, the subdirectories can be tackled one at a time.

Delivery Processes

Delivery processes are the most complex of Exim's processes. Each delivery process handles one delivery attempt for one message only, though there may be many recipients for the message and therefore many delivery actions, some of which may take place in subprocesses. Delivery processes may be started as a result of a message's arrival, by a queue runner process, or by an administrator using the *-M* option. For example:

```
exim -M 11wO3z-00042J-00
```

starts a delivery process for the given message, overriding any retry times.

When a delivery process starts up, it normally checks first to see if the message it is working on is frozen. If it is, it writes to the log:

```
Message is frozen
```

and exits without doing anything. However, a delivery process can be told to process frozen messages regardless (this happens if it is running as a result of *-M* or *-qff*, for example), and there is also an `auto_thaw` option, which automatically "thaws" (unfreezes) messages after they have been frozen for a certain length of time.

Before proceeding to the normal delivery actions, a delivery process runs the system message filter if one is configured. Exim's filtering features are described in Chapter 10, *Message Filtering*. The system filter may add or remove header lines, modify the list of recipients, or cause a message to be frozen or all its recipient addresses to be failed.

After any filtering, but before the process makes any deliveries, it determines what must be done for every recipient address by running the directors or routers as appropriate. This allows for optimization when more than one address is routed to the same host, and it also means that duplicate addresses generated by aliasing or forwarding can be discarded. The way that an address is processed to determine how the message should be delivered to it is described in Chapter 3; how Exim's configuration is used to control this process forms the subject matter of several other chapters.

If a message is to be delivered to more than one remote host, Exim can be configured to run several SMTP deliveries at once by setting `remote_max_parallel` to a value greater than one, for example:

```
remote_max_parallel = 5
```

When this is done, and there are multiple remote deliveries to be made, a delivery process forks several subprocesses at a time, up to the maximum specified. Otherwise it does remote deliveries serially without forking. The value of `remote_max_parallel` controls the maximum number of parallel deliveries created by a single Exim delivery process only. Because Exim has no central queue manager, there is no way of controlling the total number of simultaneous deliveries running on the local host if immediate delivery of incoming messages is configured.

Once all the delivery attempts are complete, if any of them failed outright, a bounce message is created and sent to the envelope sender address. It contains information about all the addresses that failed in this delivery run, and is created by calling Exim in a subprocess and writing to its standard input. If there is no sender address (that is, if the message that is failing to be delivered is itself a bounce message), no new bounce message can be sent. The failing message is left on the queue and frozen so that no further delivery attempts are made and the administrator's attention is drawn to it.

The final actions of the delivery process depend on whether all the message's recipients have been completely handled (either delivered or bounced). If there are no remaining addresses, all the spool files for the message are deleted, and "Completed" is written to the main log. Otherwise there are some addresses that have suffered temporary errors, or which were skipped for some reason. A warning message about the delivery delay is sent to the message's sender if appropriate (see the section "Delay Warning Messages," in Chapter 19), and the spool files are updated to record those addresses that have been delivered or bounced.

Variations on Delivery

A delivery process normally operates on all the recipient addresses in a message. On a host that is not permanently connected to the Internet, this is inappropriate; you want to deliver to the local addresses, but save the remote ones until the host is online. There is a discussion in the section "Intermittently Connected Hosts," in Chapter 12, on the use of `queue_remote_domains` and `queue_smtp_domains`, which are options that allow you to achieve this.

Sometimes it may be beneficial to specify the order in which remote deliveries take place. This is done by setting `remote_sort` to a list of domains; deliveries to those domains are then done in that order. Deliveries to domains that are not on the list take place after those that are listed, in an unpredictable order. For example, you could specify that deliveries to domains on your local network are done first by a setting such as:

```
remote_sort = *.mydomain.example
```

If a message contains a recipient that requires delivery to a slow remote host, this will not delay the deliveries to recipients on nearby hosts now.

Finally, there is an option called `hold_domains`, which specifies a list of domains that Exim is not to deliver except when a delivery is forced by an administrator. An address in these domains is deferred every time a delivery process encounters it. This feature is intended as a temporary operational measure for delaying the delivery of mail while some problem is being sorted out or some new configuration is being tested.

Summary of Message Handling Process Types

When Exim is run by executing its single binary, the type of process is controlled by the options with which it is called. The options for the four kinds of message handling process are summarized in Table 11-2.

Table 11-2. Message Handling Process Types

Option	Meaning
-bd	Daemon process, listening for SMTP, forks SMTP reception processes
-bs	SMTP reception process
from `inetd`	
-bs	Local SMTP reception process
-bS	Local batch SMTP reception process
-M	Forced delivery process for specific message
-q	Single queue runner process, starts delivery processes
-q <time>	Daemon process, starts queue runners
-t or none	Local reception process

A local reception process can be created by any other process, but creating the other kinds of process requires the caller to have Exim administration privileges (see the section "Privileged Users," in Chapter 19).

Other Types of Process

In addition to message handling, some other kinds of process are used for administering Exim or for debugging the configuration. Special configuration options are used for setting these up (for details, see Chapter 20). For example, the *-bp* option causes Exim to list the messages on its queue.

12

Delivery Errors and Retrying

This chapter is all about temporary delivery errors, and how Exim deals with them. In an ideal world, every message would either be delivered at the first attempt, or be bounced, and temporary errors would not arise. In the real world, this does not happen; hosts are down from time to time, or are not responding, and network connections fail. An MTA has to be prepared to hold on to messages for some time, while trying every now and again to deliver them. Some rules are needed for deciding how often the retrying is to occur, and when to give up because the retrying has been going on for too long.

A related topic is how to handle messages destined for hosts that are connected to the Internet only intermittently (for example, by dial-up lines). In this case, incoming messages have to be kept on some server host because they cannot be delivered immediately. Exim was not designed for this, and is not ideal for it, but because it is being used in such circumstances, the final section of this chapter discusses how it can best be configured.

Retrying After Errors

Delivering a message costs resources, so it is a good idea not to retry unreasonably often. Trying to deliver a failing message every minute for several days, for example, is not sensible. Even trying as often as every 15 minutes is wasteful over a long period. Furthermore, if one message has just suffered a temporary connection failure, immediately trying to deliver another message to the same host is also a waste of resources.

A number of MTAs use *message-based* retrying; that is, they apply a retry schedule to each message independently. This can cause hosts to be tried several times in quick succession. Exim is not like this; for failures that are not related to a specific message, it uses *host-based* retrying, which means that if a host fails, all messages that are routed to it are delayed until its next retry time arrives.

In fact, Exim normally bases these retry operations on the failing IP address, rather than the hostname. If a host has more than one IP address, each is treated independently as far as retrying is concerned. In the discussion that follows, we use the word "host" when talking about remote delivery errors to make it easier to read. It should be understood, however, that this refers to a single IP address, so that a host with several network interfaces is, in effect, treated as several independent hosts.

Information about temporary delivery failures is kept in a hints database called *retry* in the *db* subdirectory of Exim's spool directory. You can read the contents of this if you want to, using the *exim_dumpdb* or *exinext* utilities, which are described in Chapter 21, *Administering Exim*. The information includes details of the error, the time of the first failure, the time of the most recent failure, and the time before which it is not reasonable to try again.

Exim uses a set of configurable *retry rules* in the fifth section of the configuration file for deciding when next to try a failing delivery. These rules allow you to specify fixed or increasing retry intervals, or a combination of the two. Details of the rules are given later in this chapter, after the different kinds of error are described.

Remote Delivery Errors

Most, but not all, delays and retries are concerned with deliveries to remote hosts. Three different kinds of error are recognized during a remote delivery: host errors, message errors, and recipient errors.

Host Errors

A host error is not associated with a particular message, nor with a particular recipient of a message. The host errors are as follows:

- Refusal of connection to a remote host.
- Timeout of a connection attempt.
- An error code in response to setting up a connection.
- An error code in response to `HELO` or `EHLO`.
- Loss of connection at any time, except after the final dot that ends a message.
- I/O errors at any time.
- Timeout during the SMTP session, other than in response to `MAIL`, `RCPT` or the dot at the end of the data.

When a permanent SMTP error code (`5`*xx*) is given at the start of a connection or in response to a `HELO` or `EHLO` command, all the addresses that are routed to the host are failed, and returned to the sender in a bounce message.

The other kinds of host error are treated as temporary, and they cause all addresses routed to the host to be deferred. Retry data is created for the host, and it is not tried again, for any message, until its retry time arrives. If the current set of addresses are not all delivered to some backup host by this delivery process, the message is added to a list of those waiting for the failing host.* This is a hint that Exim uses if it makes a subsequent successful delivery to the host. It checks to see if there are any other messages waiting for the host, and if so, sends them down the same SMTP connection.

Message Errors

A message error is associated with a particular message when sent to a particular host, but not with a particular recipient of the message. The message errors are as follows:

- An error code in response to `MAIL`, `DATA`, or the dot that terminates the data.
- Timeout after sending `MAIL`.
- Timeout or loss of connection after the dot that terminates the data. A timeout after the `DATA` command itself is treated as a host error, as is loss of connection at any other time.

For a temporary message error, all addresses that are routed to the host are deferred. Retry data is not created for the host, but instead, a retry record for the combination of a host plus a message ID is created. The message is not added to the list of those waiting for this host. This ensures that the failing message will not be sent to this host again until the retry time arrives. However, other messages that are routed to the host are not affected, so if it is some property of the message that is causing the error, this does not stop the delivery of other mail.

If the remote host specifies support for the `SIZE` parameter in its response to `EHLO`, Exim adds `SIZE=`*nnn* to the `MAIL` command, so an overlarge message causes a permanent message error, because it arrives as a response to `MAIL`. However, when `SIZE` is not in use, some hosts respond to unacceptably large messages by just dropping the connection. This leads to a temporary message error if it is detected after the whole message has been sent. Better behaved hosts give a permanent

* Strictly, the list is of messages routed through the current transport that are waiting for the specific host, but in the great majority of configurations, there is usually only one **smtp** transport.

error return after the end of the message; this allows the message to be bounced without retries.

Recipient Errors

A recipient error is associated with a particular recipient of a message. The recipient errors are as follows:

- An error code in response to `RCPT`
- Timeout after `RCPT`

For temporary recipient errors, the failing address is deferred, and routing retry data is created for it. This delays processing of the address in subsequent queue runs, until its routing retry time arrives. The delay applies to all messages, but because it operates only in queue runs, one attempt is made to deliver a new message to the failing address before the delay starts to operate. This ensures that, if the failure is really related to the message rather than the recipient ("message too big for this recipient" is a possible example), other messages have a chance of being delivered. If a delivery to the address does succeed, the retry information is cleared, after which all stuck messages are tried again.

The message is not added to the list of those waiting for this host. Use of the host for other recipient addresses is unaffected, and except in the case of a timeout, other recipients are processed independently, and may be successfully delivered in the current SMTP session. After a timeout, it is, of course, impossible to proceed with the session, so all addresses are deferred. However, those other than the one that failed do not suffer any subsequent retry delays. Therefore, if one recipient is causing trouble, the others have a chance of getting through when a subsequent delivery attempt occurs before the failing recipient's retry time.

Problems of Error Classification

Some hosts have been observed to give temporary error responses to every `MAIL` command at certain times ("insufficient space" has been seen). These are treated as message errors. It would be nice if such circumstances could be recognized instead as host errors, and retry data for the host itself created, but this is not possible within the current Exim design. What actually happens is that retry data for every (host, message) combination is created.

The reason that timeouts after `MAIL` and `RCPT` are treated specially is that these can sometimes arise as a result of the remote host's verification procedures taking a very long time. Exim makes this assumption, and treats them as if a temporary error response had been received. A timeout after the final dot is treated specially because it is known that some broken implementations fail to recognize the end of the message if the last character of the last line is a binary zero. Thus, is it helpful to treat this case as a message error.

Timeouts at other times are treated as host errors, assuming a problem with the host, or the connection to it. If a timeout after `MAIL`, `RCPT`, or the final dot is really a connection problem, the assumption is that at the next try, the timeout is likely to occur at some other point in the dialog, causing it to be treated as a host error.

There is experimental evidence that some MTAs drop the connection after the terminating dot if they do not like the contents of the message for some reason. This is in contravention of the RFC, which indicates that a `5`*`xx`* response should be given. That is why Exim treats this case as a message error rather than a host error, in order not to delay other messages to the same host.

Delivery to Multiple Hosts

In all cases of temporary delivery error, if there are other hosts (or IP addresses) available for the current set of addresses (for example, from multiple MX records), they are tried in this run for any undelivered addresses, subject of course to their own retry data. This means that newly created recipient error retry data does not affect the current delivery process; instead, it takes effect the next time a delivery process for the message is run.

Local Delivery Errors

Remote deliveries are not the only cases where a temporary error may be encountered; they can also arise during local deliveries. The two most common cases are as follows:

- A delivery to a mailbox file fails because the user is over quota.
- A delivery to a command via a pipe fails, with the command yielding a return code that is defined as "temporary" (see the section "The pipe Transport" in Chapter 9, *The Transports*).

The mechanism for computing retry times is the same as for remote delivery errors, but they apply only to deliveries in queue runs. When a delivery is not part of a queue run (typically an immediate delivery on receipt of a message), the directors are always run for local addresses, and local deliveries are always attempted, even if retry times are set for them. This makes for better behavior if one particular message is causing problems (for example, causing quota overflow, or provoking an error in a filter file). If such a delivery suffers a temporary failure, the retry data is updated as normal, and subsequent delivery attempts from queue runs occur only when the retry time for the local address is reached.

Routing and Directing Errors

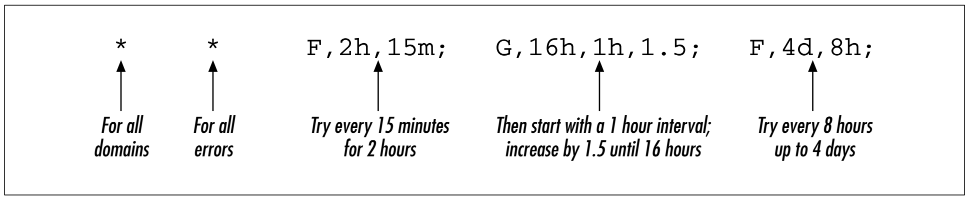
Temporary errors are also possible during routing and directing. They are most commonly caused by:

- A problem with a DNS lookup: either a timeout or a "try again" DNS error. A name server may be down for some reason, or it may be unreachable owing to a network problem. Mistakes in zone files also sometimes cause name servers to issue temporary errors.
- A problem with a lookup in a local database. The database server may be down, or, if it is on some other host, it may be unreachable.
- Mistakes in Exim's configuration (for example, a syntax error in an expansion string).

Retry processing applies to directing and routing as well as to delivering, but in the case of directing, only for delivery processes started in queue runs (as explained in the previous section). The retry rules do not distinguish between these three actions, so it is not possible, for example, to specify different behavior for failures to route the domain *snark.example* and failures to deliver to the host *snark.example*. However, although they share the same retry rule, the actual retry times for routing, directing, and transporting a given domain are maintained independently.

Retry Rules

The rules for controlling how often Exim retries a temporarily failing address are contained in the fifth part of the configuration file, following the specification of the routers. Each retry rule occupies one line and consists of three parts: a pattern, an error name, and a list of retry parameters. Figure 12-1 shows the single rule that is provided in the default configuration file. Many installations operate with just this default rule.



```
*     *     F,2h,15m;   G,16h,1h,1.5;   F,4d,8h;
```

For all domains
For all errors
Try every 15 minutes for 2 hours
Then start with a 1 hour interval; increase by 1.5 until 16 hours
Try every 8 hours up to 4 days

Figure 12-1. Default retry rule

Retry Rule Patterns

The pattern that starts a retry rule may be a plain domain, a wildcarded domain (that is, starting with an asterisk), a domain lookup (as in a domain list), a regular expression, or a complete address (*local_part@domain*). The last form must be used only with local domains.

When Exim needs to calculate when next to try to deliver a particular address, it searches the retry rules in order, and uses the first one it encounters that matches certain criteria, depending on the particular error that was encountered. If no suitable rule is found, a temporary error is converted into a permanent error, and the address is bounced after the first delivery attempt. Therefore, the final rule should normally contain asterisks in the first two fields, as shown in Figure 12-1, so that it will apply to all cases that are not covered by earlier rules.

For remote domains, when looking for a retry rule after a routing attempt has failed (for example, after a DNS timeout), each line in the retry configuration is tested only against the domain in the address. However, when looking for a retry rule after a remote delivery attempt has failed (for example, a connection timeout), each line in the retry configuration is first tested against the remote hostname, and then against the domain name in the address. For example, if the MX records for *a.b.c.d* are:

```
a.b.c.d.  MX  5  x.y.z.
          MX  6  p.q.r.
          MX  7  m.n.o.
```

and the retry rules are:

```
p.q.r    *      F,24h,30m;
a.b.c.d  *      F,4d,45m;
```

then failures to deliver the address *xyz@a.b.c.d* to the host *p.q.r* use the first rule to determine retry times, but for all the other hosts for the domain *a.b.c.d*, the second rule is used. The second rule is also used if routing to *a.b.c.d* suffers a temporary failure.

A single remote domain may have a number of hosts associated with it, and each host may have more than one IP address. Retry algorithms are selected on the basis of the domain name, but are applied to each IP address independently. If, for example, a host has two IP addresses and one is broken, Exim will generate retry times for that IP address, and will not try to use it until its next retry time comes. Thus, the good IP address is likely to be tried first most of the time.

For local domains, after a directing or a local delivery failure, the complete address (*user@domain*) is matched against retry rules that start with regular expressions or patterns containing local parts. However, if there is no local part in a pattern that is not a regular expression, the local part of the address is not used in the matching. Thus, an entry such as:

```
lookingglass.example        *  F,24h,30m;
```

matches any address whose domain is *lookingglass.example*, whether it is a local or a remote domain, whereas:

```
alice@lookingglass.example  *  F,24h,30m;
```

can be specified only if *lookingglass.example* is a local domain; it applies to temporary failures that involve the local part *alice*, but not to addresses in that domain that contain other local parts.

If local delivery is being used to collect messages for onward transmission by some other means (for example, as batched SMTP), a temporary failure may not be dependent on the local part at all. For example, the file in which all messages for a particular domain are being collected may have reached a quota limit. Both the **appendfile** and **pipe** transports have an option called `retry_use_local_part`, which can be set to false in order to suppress the inclusion of local parts when matching retry patterns. When this option is set, patterns containing local parts are skipped, and regular expressions are matched against the domain only.

Retry Rule Error Names

The second field in a retry rule is the name of a particular error, or an asterisk, which matches any error. The errors that can be specified are listed in Table 12-1.

Table 12-1. Error Field in Retry Rules

Error	Meaning
`quota`	Quota exceeded in local delivery
`quota_`*`time`*	Quota exceeded in local delivery, and the mailbox has not been read for *`time`*
`refused_MX`	Connection refused from a host obtained from an MX record
`refused_A`	Connection refused from a host not obtained from an MX record
`refused`	Any connection refusal
`timeout_connect`	Connection timed out
`timeout_DNS`	DNS lookup timed out
`timeout`	Any timeout

This field makes it possible to apply different retry algorithms to different kinds of errors; some examples are shown in the following section. The quota errors apply both to system-enforced quotas and to Exim's own quota mechanism in the **appendfile** transport.

Retry Rule Parameter Sets

The remainder of a retry rule is a sequence of retry parameter sets, separated by semicolons and optional whitespace. For example:

```
F,3h,10m; G,16h,40m,1.5; F,4d,6h
```

Each set consists of:

```
letter,cutoff time,arguments;
```

For example, in:

```
F,8h,15m;
```

the letter is F, the cutoff time is 8 hours, and there is just one argument. The letter identifies the algorithm for computing a new retry time; the cutoff time is the time beyond which this algorithm no longer applies, and the arguments vary the algorithm's action. There are two available algorithms:

- The letter `F` specifies retrying at fixed intervals. There is a single argument, which specifies the interval. In the previous example, `15m` sets up retries every 15 minutes.
- The letter `G` specifies retrying at increasing intervals.[*] The first argument specifies a starting value for the interval, and the second a multiplier. For example:

  ```
  G,16h,40m,1.5;
  ```

  specifies an initial interval of 40 minutes, which is increased each time by a factor of 1.5. The actual intervals used are therefore 40 minutes, 60 minutes, 90 minutes, and so on.

A retry rule may contain any number of parameter sets of either type, in any order. If none are provided, no retrying is done for addresses and errors that match the rule, thereby immediately turning temporary errors into permanent errors.

Computing Retry Times

When Exim computes a retry time from a retry rule, the parameter sets are scanned from left to right until one whose cutoff time has not yet passed is reached. The cutoff time is measured from the time that the first failure for the host or domain (combined with the local part if relevant) was detected, not from the time the message was received.

This set of parameters is then used to compute a new retry time that is later than the current time. In the case of fixed interval retries, this just means adding the

[*] `G` was chosen because it is next to `F`, and because the intervals form a geometric progression.

interval to the current time. For geometrically increasing intervals, retry intervals are computed from the rule's parameters until one that is greater than the previous interval is found. Consider the rule in the default configuration:

```
    *  *  F,2h,15m; G,16h,1h,1.5; F,4d,8h;
```

For the first 2 hours after a failure is detected, the next retry time is computed as 15 minutes after the most recent failure. After that, there is an interval of 1 hour before the next retry, and this is increased by a factor of 1.5 each time, until 16 hours have passed since the first failure. Thereafter, retries happen every 8 hours, until 4 days have passed.

At this point, Exim has run out of retry algorithms for the address. In this state, if any delivery suffers a temporary failure, it is converted into a permanent timeout failure, and the address is bounced. What happens if a new message for the same address arrives is described in the section "Long-Term Failures," later in this chapter.

Because a geometric retry rule might "run away" and generate enormously long retry intervals, there is a configuration option called `retry_interval_max`, which limits the maximum interval between retries. Its default value is `24h`, ensuring that all temporarily failing addresses are tried at least once a day.

Using Retry Times

The retry times that are computed from the retry rules are hints rather than promises. Exim does not make any attempt to run deliveries exactly at the computed times. Instead, a queue runner process starts delivery processes for delayed messages periodically, and these processes attempt new deliveries only for those addresses that have passed their next retry time. If a new message arrives for an address that earlier had a temporary failure, an immediate delivery is attempted if the address is local, but for a remote address, it occurs only if the retry time for that address has been reached. A continual stream of messages for a broken host does not therefore cause a continual sequence of delivery attempts.

If no new messages for a failing address arrive, the minimum time between retries is the interval between queue runner processes. There is not much point in setting retry times of 5 minutes if your queue runners happen only once an hour, unless there are a significant number of incoming messages (which might be the case on a system that is sending everything to a smart host, for example).

Retry Rule Examples

Here are some example retry rules suitable for use when *wonderland.example* is a local domain:

```
alice@wonderland.example quota     F,7d,3h
wonderland.example       quota_5d
wonderland.example       *         F,1h,15m; G,2d,1h,2;
lookingglass.example     *         F,24h,30m;
*                        refused_A F,12h,20m;
*                        *         F,2h,15m; G,16h,1h,1.5; F,5d,8h;
```

The first rule sets up special handling for mail to *alice@wonderland.example* when there is an over-quota error. Retries continue every 3 hours for 7 days. The second rule handles over-quota errors for other local parts at *wonderland.example* in the case when the mailbox has not been read for 5 days. The absence of a local part in the pattern has the same effect as supplying `*@`. As no retry algorithms are supplied, messages that fail for quota reasons are bounced immediately if the mailbox has not been read for at least 5 days. If the mailbox has been read within the last 5 days, this rule does not apply, and the next rule is used instead.

The third rule handles all other errors for *wonderland.example*; retries happen every 15 minutes for an hour, then with geometrically increasing intervals until 2 days have passed since a delivery first failed.

The fourth rule controls retries for the domain *lookingglass.example*, whether it is local or remote, and the remaining two rules handle all other domains, with special action for connection refusal from hosts that were not obtained from an MX record. The "connection refused" error means that a host is up and running, but is not listening on the SMTP port. This state exists for a short while when a host is restarting, but if it continues for some time, it is increasingly likely that the host is a workstation whose name has crept into an email address in error, because it is never going to accept SMTP connections. It therefore makes sense to bounce such addresses more quickly than normal.

The final rule in a retry configuration should always have asterisks in the first two fields so as to provide a general catchall for any addresses and errors that do not have their own special handling, unless, of course, you want such addresses never to be retried. This example tries every 15 minutes for 2 hours, then at intervals starting at 1 hour and increasing by a factor of 1.5 up to 16 hours, then every 8 hours up to 5 days.

Timeout of Retry Data

One problem with Exim's use of a host-based rather than a message-based retrying scheme arises when a host is tried only infrequently. A common example is an MX secondary host for some domain. Suppose there is a period of network failure, such that both the primary and the secondary host for a domain are unreachable. Retry data for both of them is computed. When the network comes back, mail is delivered to the primary server, leaving the retry information for the secondary still set. It could be weeks or months before the secondary is tried again. If it then fails, Exim could be in danger of concluding that it had been down all that time.

To circumvent this problem, Exim timestamps the data that it writes to its retry hints database. When it consults the data during a delivery, it ignores any that is older than the value set in `retry_data_expire` (default 7 days). If, for example, a host hasn't been tried for 7 days, Exim will try it immediately when a message for it arrives, and if that fails, it will calculate a retry time as if it were failing for the first time.

If a host really is permanently dead, this behavior causes a burst of retries every now and again, but only if messages routed to it are rare. If there is a message at least once every 7 days, the retry data never expires.

Long-Term Failures

Special processing happens when an address has been failing for so long that the cutoff time for the last algorithm has been reached. This is independent of how long any specific message has been failing; it is the length of continuous failure for the address that counts. For routing, directing, or local deliveries, a subsequent delivery failure causes Exim to time out the address, and it is bounced. For remote deliveries, it is a bit more complicated, because a remote domain may route to more than one host, each of which may have more than one IP address. The address is timed out only when the cutoff times for all the IP addresses have been passed. For example, if the domain *lookingglass.example* is routed by MX records to both *tweedledum.example* and *tweedledee.example*, and the retry rules are:

```
tweedledum.example  *  F,1d,30m;
tweedledee.example  *  F,5d,2h;
```

then the address *alice@lookingglass.example* does not time out until *tweedledum.example* has been down for more than one day and *tweedledee.example* has been down for more than five days.

Suppose there are a number of messages on the queue that are waiting to deliver to the same address, and that eventually the address times out when one message attempts a delivery. What should happen to the others? Should further deliveries be tried, or should the addresses be bounced without trying to deliver? What should happen when new messages arrive for the timed out address?

One possibility is to try a delivery for each message, although this could result in a lot of failed delivery attempts. Local deliveries do not use many resources, so Exim always tries a local delivery, even when the address timed out earlier. If the delivery fails, the address is bounced.

Remote deliveries are handled differently in order to avoid making too many potentially expensive delivery attempts. For every IP address that has passed its cutoff time, Exim continues to compute retry times, based on the final retry algorithm. Until the post-cutoff retry time for one of the IP addresses is reached, the failing address is bounced without actually trying to deliver to a remote host. This means that a new message can arrive and be bounced without any delivery attempt taking place at all. In effect, Exim is saying, "This host has been dead for five days and I tried it recently, so it is not worth trying again yet." If a new message arrives after the retry time for at least one of the IP addresses, one new delivery attempt is made to those IP addresses that are past their retry times, and if that still fails, the address is bounced, and new retry times are computed.

The final interval in a retry rule is often quite long (in the default rule, it is 8 hours). If you feel that this is too long to wait between retries of a failed host, you can add an additional parameter set specifically to shorten this time. Consider, for example, the default rule with one additional parameter set:

```
*  *  F,2h,15m; G,16h,1h,1.5; F,4d,8h; F,4d1h,1h;
```

This rule carries on trying for 4 days and 1 hour, instead of 4 days. That in itself doesn't make much difference, of course, but when this rule times out, the final retry interval is 1 hour instead of 8. This means that from then on Exim tries to deliver to a host if at least an hour has elapsed since the last failure. If less than an hour has passed, it will bounce an address without trying a delivery.

A similar kind of behavior can be specified in a different way, by setting:

```
delay_after_cutoff = false
```

in an **smtp** transport. When `delay_after_cutoff` is false, if all the IP addresses to which a domain routes are past their final cutoff time, Exim tries to deliver to those IP addresses that have not been tried since the message arrived. If there are none, or if they all fail, the address is bounced. In other words, it does not delay when a new message arrives, but tries the expired IP addresses immediately, unless they have been tried since the message arrived (presumably in delivering some other message). If there is a continuous stream of messages for the failing domains, unsetting `delay_after_cutoff` means that there will be many more attempts to deliver to failing IP addresses than in the default case when `delay_after_cutoff` is true. However, if a clump of messages arrive more or less simultaneously, some of them may be bounced without a delivery attempt.

Ultimate Address Timeout

The retry rules we have been describing work well in most normal circumstances, but there is one case where messages can be left on the queue for extended periods, potentially leading to an ever-increasing queue. This is the case where a host is intermittently available, or when a message has some attribute that prevents its delivery when others to the same address get through.

For example, if the destination's connection to the Internet is of poor quality and suffers frequent failures, short messages might be deliverable, while longer ones almost always fail. Whenever a message is successfully delivered, the "retry clock" for the host is restarted, and so it never times out. As a result, the failing messages could remain on the queue for ever. To prevent this, Exim uses another rule, called the *ultimate address timeout*, which has two parts:

- If a message has been on the queue for longer than the cutoff time of every applicable retry rule for an address (if more than one host is involved, there may be more than one retry rule), a delivery is attempted for that address to all possible hosts, even if the normal retry time has not been reached.
- After any temporary delivery failure, if the message has been on the queue for longer than the cutoff time of every applicable retry rule for the address, the address is timed out, even if there is an unexpired retry rule. New retry times are not computed in this case.

Recall an earlier example where the domain *lookingglass.example* was routed by MX records to two hosts, whose retry times were set by these rules:

```
tweedledum.example  *  F,1d,30m;
tweedledee.example  *  F,5d,2h;
```

Using this configuration, if Exim finds that a message addressed to the *lookingglass.example* domain has been on its queue for more than five days, it forces a delivery attempt for that address, regardless of the current retry times. If the delivery fails, it bounces the address, whatever the retry state of the two hosts.

Intermittently Connected Hosts

Exim was designed for running in an environment where all hosts are permanently connected to the Internet. However, hosts that use dial-up to connect to the Internet only intermittently have become quite common, because this way of working is cheaper. There are also hosts with ISDN connections whose administrators want to control when ISDN is used. For example, they do not want a connection to be made every time a local user submits a message for a remote destination.

If Exim is run in such an environment, the retrying mechanisms are not really adequate. Despite this, people are running Exim, both on intermittently connected hosts and on the servers that support them. The following sections contains some guidance on how to do this, but do bear in mind that it is stretching the purpose for which Exim was designed.

Incoming Mail for an Intermittently Connected Host

Incoming mail for intermittently connected hosts accumulates on a server that is permanently connected, and the client host collects the mail when it connects. There are two ways in which this can be done:

- Mail for each connecting host is delivered into files on the server. Sometimes a single mailbox is used for each host, independent of the local parts in the addresses. In other configurations, each message is delivered into a separate file. There are some examples of this in the section "Batched Delivery and BSMTP," in Chapter 9. The MTA on the server is not involved in storing the mail, because as far as it is concerned, the messages have been delivered. Therefore, no special attention to the retry rules is needed.
- Mail remains in the MTA's queue and is delivered using SMTP when the client connects. This requires special configuration for Exim to avoid too much pointless retrying.

If you are using Exim on the server, the first method is strongly recommended, especially if there are likely to be more than a trivial number of messages waiting for a client to connect. Exim's simple queuing strategy is based on the premise that most messages can be delivered quickly, and that therefore queue sizes will normally be small. Scanning large queues can slow things down a lot. Furthermore, if you use the second approach, you are mixing up two kinds of message in the queue: those that are having delivery problems, and those that are waiting for a client host to connect. This makes administration harder.

If you do use Exim's queue as a place to hold messages for dial-up clients (and despite the earlier remarks, this is not unreasonable if, say, there is just one client that receives a handful of messages a day), there are some configuration options that can improve its performance. You should set a long retry period for the intermittent hosts. For example:

```
cheshire.wonderland.example    *   F,5d,24h
```

This stops a lot of failed delivery attempts from occurring, but Exim remembers which messages it has queued up for that host. Once the client comes online, delivery of one message can be forced (either by using the *-M* or *-R* options, or by using the `ETRN` SMTP command), which causes all the queued up messages to be delivered, often down a single SMTP connection. While the host remains connected, any new messages are delivered immediately.

If the connecting hosts do not have fixed IP addresses (that is, if a host is issued with a different IP address each time it connects), Exim's retry mechanisms on the holding host become confused because the IP address is normally used as part of the key string for holding retry information. This can be avoided by setting `no_retry_include_ip_address` on the **smtp** transport. When this is done, the retrying is based only on the hostname. This has disadvantages for permanently connected hosts that have more than one IP address, so it is best to arrange a separate transport for these intermittently connected hosts. For example, a configuration could have two **smtp** transports, like this:

```
normal_smtp:
  driver = smtp

special_smtp:
  driver = smtp
  no_retry_include_ip_address
```

and could either use two routers for handling the different kinds of host, or a single router with a `transport` setting that selects the appropriate transport, like this:

```
lookuphost:
  driver = lookuphost
  transport = ${if match{$domain}\
              {\\.variable\\.example\$}\
              {special_smtp}{normal_smtp}}
```

This does conventional DNS routing, but selects the **special_smtp** transport for domains whose names end with *.variable.example*, and selects the **normal_smtp** transport for all others.

Exim on an Intermittently Connected Host

On an intermittently connected client host, Exim needs to be configured so that local deliveries take place immediately, but deliveries to remote domains are not attempted until a queue run is explicitly started. None of the retrying mechanism is relevant. The simplest configuration is to set `queue_remote_domains`, which provides a list of domains for which immediate delivery is not to be done. For example, on a single host that is not part of a local network, the setting should be:

```
queue_remote_domains = *
```

so that all remote domains are queued. If there are several hosts on a local network that exchange mail among themselves, queue_remote_domains can be set to exclude their domains from queuing:

```
queue_remote_domains = ! *.local.hosts
```

When a connection to the Internet is made, if a queue run is started by obeying:

```
exim -qf
```

each message is likely to be sent in a separate SMTP session, because no routing was previously done. (It is best to use *-qf* instead of just *-q*, in case there were any earlier failed delivery attempts, because *-qf* overrides retry times.) Messages from intermittently connected hosts are often all sent to a single smart host, and it is more efficient if they can all be sent in a single SMTP connection. This can be arranged by running the queue with:

```
exim -qqf
```

instead. In this case, the queue is scanned twice. In the first pass, routing is done, but no deliveries take place. It is as if every remote delivery suffered a temporary failure, and Exim updates its hints file that contains a list of which messages are waiting for which host. The second pass is a normal queue run; since all the messages have been routed earlier, those destined for the same host are likely to be sent as multiple deliveries in a single SMTP connection.

Another way of arranging for remote routing to be done in advance is to use queue_smtp_domains instead of queue_remote_domains. You can do this only if it is possible to route the remote addresses when the client is not connected to the Internet. For example, if you want to send everything to a single smart host, whose IP address you know, you can use a router such as:

```
remotes:
  driver = domainlist
  route_list = * 192.168.4.5 byname
```

where the IP address of the smart host is given explicitly. Unless you put the smart host in your */etc/hosts* file, giving its name instead of its IP address would cause a DNS lookup that would not work when the client was offline. The difference from queue_remote_domains is that Exim does the routing when the message arrives, so the queue run can be done with *-qf* instead of *-qff*, which is faster (though marginally so if there are only a few messages).

The daemon on an intermittently connected host

If you run an Exim daemon on an intermittently connected host, it should not be configured to start up any queue runner processes. That is, it should be started with just the *-bd* option. You may want to do this if you have user agents that use

SMTP to transfer messages to the MTA via the loopback interface, or if you have incoming mail from other hosts on a local network.

Incoming mail on an intermittently connected host

If incoming mail from the Internet is received using SMTP, the server is likely to send many messages over a single connection. The default behavior of Exim is to suspend automatic delivery of messages after a certain number have been received in one connection to prevent too many local delivery processes being started by a single remote host. This is not as relevant on a host that receives mail from just one source, so the value of `smtp_accept_queue_per_connection` should be increased or even set to zero (that is, disabled), so that all incoming messages down a single connection are delivered immediately.

13

Message Reception and Policy Controls

Once upon a time, when the Internet was young and innocent, MTAs accepted any message that was sent to them and did their best to deliver it, including sending it on to another host. This process is known as *relaying*. This cooperative approach has been so much abused in recent times that it is now viewed as a bad thing for an MTA to be unselective in the messages it is prepared to accept.

A host that accepts arbitrary messages for relaying is called an *open relay*; such hosts used to be common, but as levels of abuse have risen, they have almost all been eliminated. In today's Internet, hosts that relay mail must ensure that they do so only in the specific cases they are expecting to handle, for example, only relaying to certain domains or from certain hosts.

The general increase in unsolicited mail has also caused MTAs to tighten up on their controls on all incoming messages, even when relaying is not involved. Much junk mail arrives with bogus sender addresses or syntactically invalid header lines; such mail can be kept out by checking before accepting messages. No such checks can be perfect, because a clever forger can usually find a way round them, but they do reduce the size of the problem.

Exim contains a number of different controls that are specified as options in the main section of the runtime configuration file. It is configured not to do any relaying "out of the box," but other checks must be configured by the administrator if they are required.

This chapter is concerned with the way in which Exim accepts incoming messages, including the checks that may be applied during the reception process. It also covers changes that can be made to messages at the time of their reception, other than address rewriting, which is covered in the following chapter.

Message Sources

Messages received over TCP/IP are treated differently from those that are received from local processes directly, and we point out some of these differences shortly. A local process can, of course, make a TCP/IP connection to the local host, either using the loopback interface or using the host's external IP address. Messages received over such connections are treated in the same way as those that are received from remote hosts. In particular, note that the loopback address is not treated as special. If you want, for example, to allow relaying for messages received on the loopback interface, you have to configure this explicitly.

SMTP is the only way of transferring a message over a TCP/IP connection, but when a message is passed from a local process without using TCP/IP, several formats are supported, including the use of SMTP over a pipe connection. If SMTP is used in this way, the message is still treated as originating from a local user process. Only input over TCP/IP is considered "remote."

Unlike some other MTAs, Exim never changes the bodies of messages in any way. In particular, it does not attempt to convert one form of encoding into another. It is "8-bit clean," which means that the only characters it treats specially are those that it is required to interpret, such as CR (carriage return) and LF (linefeed). All other character values are treated as data.

Message Size Control

It is a good idea to set a limit to the size of message that Exim will handle. The `message_size_limit` option controls this limit; a value of zero (the default) means no limit. For example:

```
    message_size_limit = 12M
```

sets a limit of 12 MB. Messages that are larger than the limit are not accepted.*

When Exim creates a bounce message, it appends the original message to the end of the error message text. To avoid sending excessively large bounce messages, there is a limit to the amount of original message that is copied. This is set by `return_size_limit`, which defaults to 100 KB. The body of the original message is truncated when the limit is reached,† and a comment pointing this out is added at the top. The value of `return_size_limit` should always be somewhat smaller than `message_size_limit`.

* There is a transport option, also called `message_size_limit`, which limits the size of message a particular transport will handle. To have any effect, this must, of course, be less than the global limit.

† The limit is not exact to the last byte, owing to the use of buffering for transferring the message in chunks.

Messages from Local Processes

Messages received directly from local processes are not subject to any of the verification and other checks that are used for messages arriving over TCP/IP.

An Exim reception process may be started by any locally running process. Most commonly, this happens when a user instructs a user agent to send a message, but other processes are also able to send messages if they wish. For example, if a command that is automatically run by *cron* produces output, it is mailed to the user. For historical reasons, processes that send messages in this way call the local MTA using one of the paths */usr/sbin/sendmail* or */usr/lib/sendmail*. Whichever of these paths is conventional in your operating system should be set up as a symbolic link to Exim, which is compatible with the Sendmail interface for accepting messages in this way.

Exim can be run directly from a shell, passing options and arguments on the command line, and the message on the standard input, but this is generally useful only for testing because you have to supply the entire message, including all the header lines. For sending "real" messages from a shell, it is better to use a user agent such as the Unix *mail* command.

Addresses in Header Lines

In a locally submitted message, if an unqualified address is found in any of the header lines that contain addresses, it is qualified using the domain defined by `qualify_domain` (for senders) or `qualify_recipient` (for recipients) at the time the message is received. For example, on a host called *ahost.plc.example*, you might have configured:

```
qualify_domain = plc.example
qualify_recipient = ahost.plc.example
```

If an incoming message contained:

```
From: theboss
To: thedogsbody
```

Exim would convert these lines into:

```
From: theboss@plc.example
To: thedogsbody@ahost.plc.example
```

Specifying Recipient Addresses

There are a number of different ways of passing recipient envelope addresses to Exim from a local process. In all cases, unqualified addresses are permitted; they are qualified using the domain defined by the `qualify_recipient` option. This defaults to the value of `qualify_domain`, which in turn defaults to the name of the local host.

- If none of *-bs*, *-bS*, or *-t* are present as options, the recipients are passed as the command's arguments. Each argument may be a comma-separated list of RFC 822 addresses. In other words, if Exim is called directly from a shell, either commas or spaces can be used to separate the addresses. For example, one could type:

  ```
  exim  user1@example.com,user2@another.example.com  user3
  ```

  followed by the message, complete with RFC 822 header lines. The message is terminated by an end-of-file indication, or, unless the *-oi* option is given, by a line consisting of a single dot character.

- If the *-t* option is present on the command line, no arguments are normally given. Exim constructs the list of envelope recipients by extracting all the addresses from any *To:*, *Cc:*, and *Bcc:* header lines within the message. Then it removes any *Bcc:* header lines. This is the only case where *Bcc:* lines are removed from messages; if they are present in messages received in any other way, they are preserved.

  If the command does have arguments, they are interpreted as addresses *not* to send to; in other words, any argument addresses that also appear in the header lines are ignored. This action, removing any argument addresses from the recipients list, accords with published Sendmail documentation, but it appears that some versions of Sendmail *add* addresses given on the command line, instead of removing them. Exim can be made to behave in this fashion by setting:

  ```
  extract_addresses_remove_arguments = false
  ```

- If the *-bs* option is present on the command line, Exim expects to receive SMTP commands on its standard input, and it writes responses to the standard output.*

- If the *-bS* option is present on the command line, Exim expects to receive SMTP commands on its standard input, but it does not write any responses. This is so-called *batch SMTP*, where the SMTP commands are just being used as a convenient way of encoding the envelope addresses.

For both kinds of SMTP, more than one message can be passed in one connection, whereas the other cases are limited to one message at a time. Further details about SMTP handling are given in Chapter 15, *Authentication, Encryption, and Other*

* This option is also used when running Exim under *inetd*; Exim can tell the difference, because, in this case, the standard input is a socket with an associated remote IP address. When started from *inetd*, Exim treats the message as coming from a remote host.

SMTP Processing. The sources of recipient addresses and the command-line options that affect them are summarized in Table 13-1.

Table 13-1. Recipient Address Sources

Option	Meaning
none	Command arguments are recipients
-t	Recipients from headers (`Bcc:` removed)
-bs	Recipients from local SMTP RCPT commands
-bS	Recipients from batch SMTP RCPT commands

Local Sender Addresses

The source of the envelope sender address for messages submitted locally (that is, not over TCP/IP) depends on whether the calling user is trusted or not. How trusted users are defined is discussed in the section "Privileged Users," in Chapter 19, *Miscellany*. A trusted user is permitted to supply a sender address, via the `MAIL` command if the message is being received using SMTP, or otherwise in one of the following ways:

- By including a line in one of the following forms at the start of the message, preceding the RFC 822 header lines:

  ```
  From address Fri Dec 31 23:59 GMT 1999
  From address Fri, 31 Dec 99 23:59:59
  ```

  The origin of the use of a line such as this is the transfer of mail using UUCP (see RFC 976). It was then adopted as a message separator line in "Berkeley format" mailbox files, so is frequently found in saved messages. Because several different formats are encountered, Exim recognizes this line by matching it against a regular expression, which is defined by the `uucp_from_pattern` option. The default pattern matches the two formats that were previously shown, leaving the value of the address in the $1 variable.

  If `uucp_from_pattern` matches, then Exim expands the contents of `uucp_sender_address` (whose default value is `$1`) to obtain a sender address for the message. The expanded string is parsed as an RFC 822 address. For example, if the message starts with:

  ```
  From a.oakley@berlin.example Fri Jan  5 12:35 GMT 1996
  ```

  and `uucp_from_pattern` is set to its default value, expanding the contents of `uucp_sender_address` yields *a.oakley@berlin.example*. If there is no domain in the result, the local part is qualified with `qualify_domain` unless it is the empty string.

- By supplying a sender address using the *-f* command-line option. For example:

  ```
  exim -f '<f.butler@berlin.example>'
  ```

  If *-f* is present, it overrides any `From` line that may be in the message.

If the caller of Exim is not trusted, a sender address supplied in any of these ways is recognized, but ignored. Instead, Exim creates an envelope sender for the message using the login ID of the process that called it, and the domain specified by `qualify_domain`.

Unqualified Addresses from Remote Hosts

The RFCs specify that all the addresses in a message that is received from another host, both in the envelope and in the header lines, must be fully qualified; that is, they must contain both a local part and a domain. There is only one exception: the unqualified address *postmaster* is required to be accepted. Other unqualified addresses, such as those in the following SMTP commands:

```
MAIL FROM:<caesar>
RCPT TO:<brutus>
```

cause error responses because of the lack of a domain. However, when SMTP is being used as a submission protocol from local workstations, there is sometimes a need to relax this restriction, so it is possible to permit certain hosts to send messages containing unqualified addresses by setting either or both of `sender_unqualified_hosts` or `receiver_unqualified_hosts` to lists of such hosts. For example, a gateway on the local network 192.168.45.0 might permit its local clients to send unqualified addresses by setting:

```
sender_unqualified_hosts = 192.168.45.0/24
receiver_unqualified_hosts = 192.168.45.0/24
```

Unqualified addresses are not retained in the messages Exim receives. Each such address is qualified as soon as it is accepted, using the domain specified in `qualify_domain` for sender addresses, and the domain specified in `qualify_recipient` for recipients. The value of `qualify_domain` defaults to the name of the local host, and the value of `qualify_recipient` defaults to the value of `qualify_domain`.

On a host called *delta.plc.example*, if neither of these options is set, the unqualified address *apollo* becomes *apollo@delta.plc.example*, but if:

```
qualify_domain = plc.example
```

is set, it becomes *apollo@plc.example* in all cases. However, with the following:

```
qualify_domain = plc.example
qualify_recipient = delta.plc.example
```

the shorter domain is used only for sender addresses. By the time a message comes to be delivered, its envelope contains only fully qualified addresses.

If there are any unqualified addresses in header lines, they too are qualified, provided the message came from a host that is permitted to send unqualified addresses. If not, unqualified addresses in header lines are left untouched; they are neither qualified nor subjected to rewriting (see Chapter 14, *Rewriting Addresses*).

Checking a Remote Host

In this section, we cover a number of different checks that can be applied to a remote host when it connects to Exim in order to send a message. These checks are independent of the addresses in the message it sends and therefore apply in all cases. Checks for relaying are described in the section "Relay Checking".

Verifying a Host's Name

On receiving a TCP/IP connection, the only identification for a remote host that Exim has is its IP address. The host should give its name as the data in a `HELO` or `EHLO` command, but there is no guarantee that this is the name by which it is registered. It is particularly common for workstations that are using SMTP to submit mail to misbehave in this regard. The name supplied by `HELO` or `EHLO` is placed in the variable `$sender_helo_name`.

The only way Exim can find a properly registered name for the host is to look for a name that is associated with the IP address. This might be found in a local hosts table in */etc/hosts*, but more commonly a DNS lookup is required. This raises two problems: first, that DNS lookups can be expensive and take time, and second, that substantial numbers of Internet hosts are only registered "one way" in the DNS. That is, you can look up their names to find an IP address, but not *vice versa*.

For this reason, Exim does not try to find the registered name of a connecting host unless its configuration requires this to be done. If you can arrange your configuration to avoid these lookups, you will get a bit of extra efficiency. The default configuration file is designed for use on small client hosts, and it contains the setting:

```
host_lookup = *
```

This forces Exim to look up a hostname for every incoming connection. If you are running a busy server, it is probably a good idea to remove this setting, or modify it to be less general. For example, you could set something such as:

```
host_lookup = 192.168.3.0/24
```

to constrain the lookup to hosts on your local network. Although `host_lookup` can contain any item that is allowed in a host list, it normally contains only IP addresses. There is not much point in including hostnames, because this implies finding a hostname in order to check whether to find a hostname!

When a hostname is found by looking up the IP address, it is placed in the variable `$sender_host_name`. If no lookup has been done, or if the lookup failed, this variable is empty. Failure to find a hostname does not by itself cause a connection to be rejected, but it might do so if a hostname is needed for checking one of the options described in the following sections. In Exim's log files, hostnames that have not been verified but are just the data from `HELO` or `EHLO` commands are shown in parentheses.

Some broken client hosts have been observed to specify the server's name, or one of its local domains, instead of their own name in `HELO` or `EHLO` commands. Suppose the host *server.plc.example* is handling the domain *plc.example*. A client host called *broken.example*, which is broken in this way, may send one of these commands:

```
ehlo server.plc.example
ehlo plc.example
```

when what it is supposed to send is:

```
ehlo broken.example
```

Using the supplied name in log lines (even in parentheses) is likely to cause confusion, so when this happens, Exim always tries to look up the correct name for the host, whatever the setting of `hosts_lookup`.

Verifying EHLO or HELO

The RFCs specifically state that mail should not be refused on the basis of the data content of the `HELO` or `EHLO` commands. However, there are installations that do want to be strict in this area, and Exim has the `helo_verify` option in order to support them. The value of this option is a list of hosts for which stricter checking is required.

When the sending host matches `helo_verify`, a `HELO` or `EHLO` command must precede any `MAIL` commands in an incoming SMTP connection. Otherwise, all `MAIL` commands are rejected with a permanent error code. In addition, the argument supplied by HELO or EHLO is verified. If it is in the form of a literal IP address in square brackets, it must match the actual IP address of the sending host. If it is a domain name, the sending host's name is looked up from its IP address and compared against it. If the lookup or the comparison fails, the IP addresses associated with the HELO or EHLO name are looked up and compared against the sending host's IP address. If none of them match, the HELO or EHLO command is rejected with a permanent error code, and an entry is written in the main and reject logs.

Even when helo_verify is not used, Exim rejects HELO and EHLO commands that contain syntax errors. One common mistake is the appearance of underscore characters in domain names.* Because this error is so very widespread, Exim permits it by default, but you can make it more strict by specifying:

```
helo_strict_syntax = true
```

This option applies only to the use of underscores. Occasionally, there is a need to allow some hosts to send real junk in a HELO or EHLO command (including no data at all); such hosts can be accommodated by setting helo_accept_junk_hosts. For example:

```
helo_accept_junk_hosts = 192.168.5.224/27
```

removes the syntax check entirely for hosts on that network.

Using a DNS Blocking List

As a service to the Internet community, a number of organizations are maintaining lists of hosts from which it might be desirable not to accept connections. These lists are maintained in the DNS, to make them easily accessable to all, and the hosts are indexed by IP address. The original list that started this trend is called the Realtime Blackhole List (RBL), and the acronym *RBL* is used in what follows as a generic term for DNS lists of this type.

The original RBL is a hand-maintained "blacklist" of hosts that, in the judgment of the list maintainers, have been misbehaving. See *http://mail-abuse.org/rbl/* for further details. Another list that is now managed by the same site is the *Dial-up User List* (DUL).† This list contains IP addresses that are used by ISPs for their dial-up customers. Many administrators configure their MTAs to refuse direct SMTP connections from such hosts, arguing that dial-up users should send out mail via their

* There is a common misapprehension that not allowing underscores is a DNS restriction. It is not; a domain name in the DNS may contain a wide variety of characters (see RFC 2181). However, there is a restriction in RFC 821, the specification of SMTP, which permits only letters, digits, dots, and hyphens in domain names that are used in SMTP transactions.

† See *http://mail-abuse.org/dul/*.

ISP's servers, where it can be better controlled. Several of the other lists use automatic means for detecting open relays; some of their policies have proved controversial.

Configuring Exim to use an RBL

You can configure Exim to make use of any number of these lists. You can also arrange for it to reject connections from hosts that are on such a list, or just log warnings, and optionally, add header lines to messages from suspect hosts. Which policy you choose depends on your circumstances and requirements. End users more often specify blocking, whereas some ISPs choose just to add header lines that their customers can inspect.

The RBL processing is done only if an incoming call is received from a host that matches `rbl_hosts`. This allows you to exclude, for example, connections from hosts on your local network, thereby saving some resources. For example:

```
rbl_hosts = ! 192.168.56.224/28
```

specifies that RBL processing is to be done only for connections from hosts that are not in the 192.168.56.224/28 network.

The default setting of `rbl_hosts` includes all hosts; however, no lookups are actually done unless `rbl_domains` is set. This lists the DNS domains in which the incoming IP address is to be looked up. For example, if the setting is:

```
rbl_domains = blackholes.mail-abuse.org:dialups.mail-abuse.org
```

and an SMTP call is received from the IP address 192.168.8.1, DNS lookups for the two domains

```
1.8.168.192.blackholes.mail-abuse.org
1.8.168.192.dialups.mail-abuse.org
```

are done. Each item in `rbl_domains` can be followed by `/warn` or `/reject` to specify what is to be done when a match is found. For example:

```
rbl_domains = blackholes.mail-abuse.org/warn : dialups.mail-abuse.org/reject
```

After matching an item with `/warn`, further items are considered, but after `/reject`, Exim stops scanning `rbl_domains`. The action for domains without either of these options is controlled by `rbl_reject_recipients`, which implies `/reject` when set (the default). If a lookup times out or otherwise fails to give a decisive answer, the mail is not blocked.

RBL warnings

The RBL warning action consists of writing a message to the main and reject logs, and if `rbl_warn_header` is true (the default), adding an *X-RBL-Warning:* header to the message. This can be detected later by system or user filters. If a host appears in several RBL lists, more than one such header may be added to a message.

RBL rejection

Rejection is done by refusing all the recipients of the message (that is, by giving permanent error returns to all `RCPT` commands), unless the message's sender is listed in `recipients_reject_except_senders`, or the recipient itself is listed in `recipients_reject_except`. It is fairly common to set:

```
recipients_reject_except = postmaster@your.domain
```

to allow your host to accept mail to the postmaster from blacklisted hosts. *X-RBL-Warning:* headers are still added to messages that get accepted as a result of an exception list.

When a message is being rejected, if a DNS TXT record associated with the host is found in the RBL list, its contents are returned as part of the 550 rejection message, unless `prohibition_message` is set (see the section "Customizing Prohibition Messages," later in this chapter). In this case, a locally specified message (possibly including the TXT data) is used.

Using RBL data values

All the host listing schemes use DNS address records with domain names constructed by reversing the IP address. The original RBL scheme completes the record with the value 127.0.0.1 on the righthand side, for example:

```
17.23.168.192.blackholes.mail-abuse.org.  A  127.0.0.1
```

The existence of that record means that the IP address 192.168.23.17 is on the RBL list; the value of the righthand side is not used for anything. However, some of the lists are now using different values to distinguish between different kinds of entry. The ORBS database,* for example, currently uses 127.0.0.2 for a confirmed open relay, 127.0.0.3 for a manual entry, and 127.0.0.4 for a block that applies to a whole network.

To allow Exim to make distinctions between these different kinds of record, a domain name in `rbl_domains` can be followed by an equals sign and a comma-separated list of IP addresses. It then acts only if a DNS record contains one of the given values. For example:

```
rbl_domains = relays.orbs.org=127.0.0.2/reject
```

applies only to ORBS entries whose righthand side is 127.0.0.2.

* See *http://www.orbs.org/.*

Remaining RBL options

There are two further options that can be added to items in `rbl_domains`:

- If `/accept` appears, a host that matches the item is accepted, and no further items in `rbl_domains` are considered. However, earlier entries may already have added warning headers. This provides a "white list" facility that can be used in conjunction with a local DNS domain to provide a list of hosts that are accepted even if they appear in subsequent blacklists.
- If `/skiprelay` appears, the use of that entry is skipped if the calling host matches `host_accept_relay`. In other words, if Exim has been explicitly configured to accept relaying requests from the calling host, RBL lookups with `/skiprelay` are omitted.

Explicit Host Blocking

There are two options for blocking incoming mail from specific hosts: `host_reject` and `host_reject_recipients`. They differ only in the way that the rejection is carried out. For hosts that match `host_reject`, an SMTP error code is returned as soon as a matching host connects, but for those that match `host_reject_recipients`, the rejection is done by rejecting every RCPT command, unless the address it contains matches `recipients_reject_except`.

When `host_reject` is used, the message remains on the remote host, which may try to deliver it to alternative MX hosts or may try to redeliver it again later. For this reason, it is better to use `host_reject_recipients`. Unless the remote host is seriously broken, a permanent rejection of every recipient causes the message to be bounced and not retried. Also, using `host_reject_recipients` allows exceptions to be made by setting `recipients_reject_except`.

The value for either of these options can contain any valid host list items. For example:

```
host_reject_recipients = ! xx.yy.zz : *.yy.zz : ! *.zz
```

rejects mail from any host outside the *zz* domain as well as all hosts in the *yy.zz* domain, except for *xx.yy.zz*. This follows from the way Exim processes host lists from left to right. If the host is *xx.yy.zz*, it matches the first item, and because this is a negative item, the host is not rejected. Any other host in the *yy.zz* domain matches the second item, which is positive, so it is rejected, whereas any other host in the *zz* domain matches the third item, so it not rejected. A host list that ends with a negative item has an implied `:*` added to it, so all hosts that are not in the *zz* domain are rejected.

The use of wildcards in host lists (such as the previous asterisk) causes a lookup of the IP address, in order to obtain the host's name so that the match can take place. If a name cannot be found, rejection occurs.* For this reason, if you have any IP addresses in the list, they should precede any names if possible, so they can be checked without needing a lookup. Otherwise, a failing lookup for an earlier item may prevent an IP address from being tested.

Checking Remote Sender Addresses

The sender address in a message's envelope is the address to which bounce messages are sent. If a host accepts a message with a bad sender address, but subsequently cannot deliver the message, the result is an undeliverable bounce message that the postmaster has to sort out. On very busy hosts, such problems are often simply ignored by setting `ignore_errmsg_errors`, but it is also reasonable to do some checking on a sender address before accepting a message.

The most common causes of bad sender addresses are as follows (not in order):

- Misconfigured mail software, frequently an MUA running on a single-user workstation, especially when this is shared between several users, so that each one has to set up their own address each time they use it.
- The use of domains that are not registered in the DNS.
- Misconfigured name servers; hostmasters are only human, and typos in zone files can arise.
- Broken gateways from other mail systems that fail to create a valid SMTP envelope sender.
- Forgery, which is very common in spam mail.

Verifying SMTP Sender Addresses

Exim does no checking of sender addresses by default, other than ensuring that they conform to the syntax required by RFC 821. If you are running a large, very busy host, you might choose to run it this way because it does not hold up the SMTP dialog while the check is done. The consequence is that you are likely to end up with significant numbers of undeliverable bounce messages, but on busy hosts these are often discarded. On most Exim hosts, however, it is probably a good idea to set the following:

```
sender_verify
```

This causes Exim to verify the sender address when it is received in the `MAIL` command. What exactly is meant by *verification*? The idea is to try to detect those sender addresses that a bounce message could not be delivered to, so what Exim

* This can be disabled by including `+allow_unknown` or `+warn_unknown` in the host list; see the section "Host Lists" in Chapter 18, *Domain, Host, and Address Lists*, for further discussion of host lists.

does is to run the address through the directors or routers, as if it were being asked to deliver a message to it.

For local addresses, this process checks the validity of the local part of the address, but for remote addresses only the domain can, in general, be verified in this way. This means that successful verification by the routers and directors cannot guarantee that an address is deliverable, but a failure to verify does guarantee that it is not deliverable.

Verify callbacks

For sites that are prepared to expend more resources on sender verification, Exim (from Release 3.20 onwards) can be configured to do some further checking after it has verified a remote address by successfully routing it. This entails making an SMTP call to one of the hosts to which the domain resolves, and testing the address as if it were the recipient of a bounce message. Exim sends:

```
HELO the local hostname
MAIL FROM:<>
RCPT TO:<the address to be tested>
QUIT
```

If the response to the `RCPT` command is a 2*xx* code, verification succeeds. If it is 5*xx*, verification fails. For anything else, and in cases when Exim cannot contact any of the relevant hosts, verification fails with a temporary error code.

This process is called *verification callback*, and it occurs only if the sending host matches `sender_verify_hosts_callback` (along with `sender_verify_hosts`), and if in addition the sender's domain matches `sender_verify_callback_domains`. Both of these options are unset by default. There is also an option called `sender_verify_callback_timeout`, which sets a timeout for connecting and for each command. It defaults to 30 seconds.

Callback verification is expensive, and is therefore not recommended for general use, especially on busy hosts. However, on a small host that handles only a few messages a day, using callback may be an acceptable overhead. It allows you to reject senders with valid domains but invalid local parts, something that is commonly encountered in spam messages. For this you would set:

```
sender_verify_hosts_callback = *
sender_verify_callback_domains = *
```

Another instance where the cost of callback might be acceptable is a corporate gateway that checks addresses in domains that are local to the corporation, but not local in the Exim sense, using a configuration such as this:

```
sender_verify_hosts_callback = *
sender_verify_callback_domains = *.plc.example
```

The assumption is that any domain matching **.plc.example* resolves to a host on the local LAN, so that making an SMTP call to it is relatively cheap.

Configuring routers and directors for verification

Without any special options, all directors and routers are used both for verification and message delivery. However, checking an address exactly as if for delivery is not always appropriate when verifying. For example, for a local address, there is little point in checking whether a user has a *.forward* file or not. The routers and directors can detect whether the address they are handling is being verified or processed for delivery, and there are a number of options that change their behavior.

If `no_verify` is set, the driver is skipped when verifying. The default configuration sets `no_verify` on the director that handles *.forward* files. Verification of senders and recipients can be separately controlled by setting:

```
no_verify_sender
no_verify_recipient
```

if necessary; `no_verify` is just a shorthand for setting both of these options. The converse option is `verify_only`; when this is set, the driver is run only when verifying, and not when delivering. The final option is `fail_verify`; if it is set and a driver accepts an address when verifying, verification fails instead of succeeding. An example of how this can be used is given in the section "Changing a Driver's Successful Outcome," in Chapter 6, *Options Common to Directors and Routers*. By setting a suitable combination of these options you could, if you wished, make verification use an entirely different set of drivers to delivery.

Exceptions to Sender Verification

Sender verification does not apply to batch SMTP input by default, but:

```
sender_verify_batch
```

can be set to cause it to happen if required. For nonbatch SMTP, verification can be constrained to a specific set of hosts by setting `sender_verify_hosts`. For example, if a cluster of hosts passes messages between themselves, having verified the senders on input, you can configure them not to waste effort on verification for messages from each other by a setting such as:

```
sender_verify_hosts = ! *.cluster.example
```

Temporary Sender Verification Failures

If a sender address cannot immediately be verified (for example, because of DNS timeouts), Exim gives a temporary failure error response (code 451) after the data for the message has been received. The error is delayed until this time instead of being given in response to the `MAIL` command, so that the message's header lines can be logged. However, if `sender_try_verify` is set, the sender is accepted with a warning message after a temporary verification failure.

Exim remembers temporary sender verification errors in a hints database. Subsequent temporary errors for the same address from the same host within 24 hours cause a 451 error after `MAIL` instead of after the data. This reduces the amount of data in the reject log and the amount repeatedly transferred over the net.

If `sender_verify_max_retry_rate` is set greater than zero, and the rate of temporary rejection of a specific incoming sender address from a specific host in units of rejections per hour exceeds it, the temporary error is converted into a permanent verification error. This should help to stop hosts hammering too frequently with temporarily failing sender addresses, which is something that certain broken hosts have been observed to do.

The default value of the option is 12, which means that a sender address that has a temporary verification error more than once every 5 minutes is soon permanently rejected. Once permanent rejection has been triggered, subsequent temporary failures all cause permanent errors, until there has been an interval of at least 24 hours since the last failure. After 24 hours, the hint expires.

Permanent Sender Verification Failures

What happens if verification fails with a permanent error depends on the setting of `sender_verify_reject`. If it is set (the default), then the message is rejected. Otherwise, if:

```
    no_sender_verify_reject
```

is set, a warning message is logged, and processing continues.

Because remote postmasters always want to see the message headers when there is a problem, Exim does not give an error response immediately if a sender address fails to verify, but instead it reads the data for the message first, and then gives a permanent error code (550) when all the data has been received. The headers of rejected messages are written to the reject log, for use in tracking down the problem or tracing mail abusers. Up to three envelope recipients are also logged with the headers.

Unfortunately, there is some software in use that treats any SMTP error response given after the data has been transmitted as a temporary failure. RFC 821 is quite clear in stating that all codes starting with 5 are always "permanent negative completion" replies. However, it does not give any guidance as to what should be done on receiving such replies, and some software persists in trying to resend messages when they receive such a code at the end of the data.

To get around this, Exim keeps a database in which it remembers the bad sender address and hostname when it rejects a message. If the same host sends the same bad sender address within 24 hours, Exim rejects the message at the `MAIL` command without reading any recipients or the data for the message. Once again, this should prevent the remote host from trying to send the message again, but there seem to be plenty of broken software out there that does keep on trying, sometimes for days on end, after receiving a negative response to a `MAIL` command.

In an attempt to shut such programs up, if the same host sends the same bad sender for a third time within 24 hours, `MAIL` is accepted, but all subsequent `RCPT` commands are rejected with a 550 error code. If the sending software does not treat that as a hard error, it is very seriously broken indeed. There is still, however, one problem that arises with this approach. On receiving Exim's 550 error response, which contains one of the messages:

```
550 unknown local part in sender
550 cannot route to sender
```

depending on whether the address was local or remote, at least one MTA insists on adding its own interpretation of error 550 as:

```
550 Unknown user
```

and this catches people's eye, causing them to ignore what Exim is trying to tell them.

This three-stage rejection process developed as a result of experience with early versions of Exim. It is now rather overcomplicated, and in a future release all rejections may be done by the method that works best: rejecting `RCPT` commands.

Fixing Bad Envelope Senders

In some cases where the sender address cannot be verified, the message itself contains a valid return address in one of its header lines. This has been noticed in messages that have arrived on the Internet through a gateway from some other mail regime. Exim can be configured to check for this case, and patch things up by replacing the broken envelope sender with a valid address from the header.

When `sender_verify_fixup` is set as well as `sender_verify`, Exim does not reject a message if the sender is invalid, provided it can find a *Sender:*, *Reply-To:*, or *From:* header containing a verifiable address. Instead, it replaces the envelope sender with the valid address, and records the fact that it has done so by adding a header of the form:

```
X-BadReturnPath: abc@bad.example rewritten
  as xyz@good.example using From header
```

If there are several occurrences of any of the relevant headers, they are all checked. If any *Resent-* header lines exist, it is those that are checked rather than the original ones.

The fixup happens for both permanent and temporary errors. This covers the case when the bad addresses refer to some DNS zone whose name servers are unreachable. This approach is, of course, fixing the symptom and not the disease, and it is not recommended for general use.

If `sender_verify_fixup` is set when `sender_verify_reject` is false, Exim does not modify the message, but records in the log the fixup it would have made.

Testing Sender Verification

You can test how Exim would respond to a request to verify a particular sender address by using the *-bvs* option:

```
exim -bvs homer@greece.example
```

There is a difference between *-bvs* and *-bv* only if your configuration distinguishes between senders and recipients when verifying, for example, by the use of `sender_verify` on some director or router.

If the configuration of your directors and routers includes any tests of values associated with a sending host, for example, checking the contents of `$sender_host_address`, you can supply values for the test using one of the command-line options whose names start with *-oM* (see the section “Remote Host Information” in Chapter 20, *Command-Line Interface to Exim*).

Checking Senders in Header Lines

Exim’s sender verification options can be used to block messages with bad envelope senders. However, recall that bounce messages are identified as such by not having an envelope sender. They are transmitted using the command:

```
MAIL FROM:<>
```

that is, the sender address is empty. Some sites have been known to block such messages on the grounds that their senders cannot be verified; this is totally misguided because it means that no legitimate bounce messages can ever be delivered to such sites, which does their users something of a disservice.

Obviously, Exim’s envelope sender checking cannot apply to an empty sender address; it must always be accepted. However, because some people feel that accepting a message without verifying *some* sender is not a good idea, an alternative check is provided.

If headers_sender_verify_errmsg is set for messages that have null senders (purporting to be bounce messages), Exim does some checking of the header lines instead. It looks for a verifiable address in the *Sender:*, *Reply-To:*, and *From:* lines. If one cannot be found, the message is rejected, unless headers_checks_fail is false, in which case it just makes a warning entry in the reject log.

If there are several occurrences of any of the relevant headers, they are all checked. If any *Resent-:* headers exist, it is those headers that are checked rather than the original ones.

Unfortunately, because it has to read the message before doing this check, the rejection happens after the end of the data, and it is known that some client programs do not treat permanent errors correctly at this point; they keep the message on their spools and try again later, but that is their problem, though it does waste some of your resources.

The option headers_sender_verify is also available. It insists on there being a verifiable address in either *Sender:*, *Reply-To:*, or *From:* on all incoming SMTP messages, not just those with null senders.

The sender_verify_hosts option applies to both of these header checking options as well as to sender_verify. The checking is done only for hosts that match it.

Explicitly Rejecting Senders

Individual outbreaks of spam often use a specific envelope sender address. If you are running a system with many mailboxes and you notice the arrival of such messages early enough, you can sometimes spare some of your users (usually those whose names start further down the alphabet) by blocking messages from the offending sender, even though it may verify successfully. There may also be other circumstances in which you want to block messages from certain senders.

There are two options for doing this—sender_reject and sender_reject_recipients—each of which contains an address list, as described in the section "Address Lists," in Chapter 18. For example:

```
sender_reject = spamuser@some.domain.example:spam.domain.example
sender_reject = partial-dbm;/etc/mail/blocked/senders
```

Like the similar options for host rejection, they differ only in the way the rejection is done. If a sender address matches sender_reject, the MAIL command is rejected with a permanent error, but as previously mentioned, this does not always cause the remote host to give up. It is normally better to use sender_reject_recipients, which accepts the MAIL command, but rejects all subsequent RCPT commands (except for any recipients listed in recipients_reject_except). The availability of an exception list is another reason for using this option.

Summary of Sender Checking Options

The options that control sender checking are summarized in this section. This checking applies only to messages that arrive over TCP/IP connections.

`headers_checks_fail` (Boolean, default = true)
  If this option is true, failure of any of the header checks causes the message to be rejected. If it is false, a warning message is written to the reject log.

`headers_sender_verify` (Boolean, default = false)
  If this option is set with `sender_verify`, and the sending host matches `sender_verify_hosts`, Exim insists on there being at least one verifiable address in the *Sender:*, *Reply-To:*, or *From:* headers (which are checked in that order) on all incoming SMTP messages. If one cannot be found, the message is rejected, unless `headers_checks_fail` is unset, in which case a warning entry is written to the reject log.

`headers_sender_verify_errmsg` (Boolean, default = false)
  This option acts like `headers_sender_verify`, except that it applies only to messages whose envelope sender is empty; that is, bounce messages.

`sender_reject` (address list, default = unset)
  This option can be set in order to reject mail from certain envelope senders. If the check fails, a 550 return code is given to `MAIL`.

`sender_reject_recipients` (address list, default = unset)
  This operates in exactly the same way as `sender_reject` except that the rejection is given in the form of a 550 error code to every `RCPT` command (unless the recipient is in `recipients_reject_except`) instead of rejecting `MAIL`.

`sender_try_verify` (Boolean, default = false)
  If this option is true, envelope sender addresses on incoming SMTP messages are checked to ensure that they are valid, but if the verification cannot be completed immediately, the message is accepted.

`sender_verify` (Boolean, default = false)
  If this option is true, envelope sender addresses on incoming SMTP messages are checked to ensure that they are valid, but if the verification cannot be completed immediately, a temporary error code is given for the `MAIL` command.

`sender_verify_batch` (Boolean, default = false)
  Sender verification is applied to batch SMTP input only if this option is set.

`sender_verify_fixup` (Boolean, default = false)
  If `sender_verify` and `sender_verify_reject` are true and this option is also true, an invalid envelope sender or one that cannot immediately be verified is replaced by a valid value from a header line. If `sender_verify_reject` is false,

the envelope sender is not changed, but Exim writes a log entry giving the correction it would have made.

`sender_verify_hosts` (host list, default = `*`)
: If `sender_verify` or `sender_try_verify` is true, this option specifies a list of hosts to which sender verification applies. The check caused by `headers_sender_verify` also happens only for matching hosts.

`sender_verify_max_retry_rate` (integer, default = 12)
: If this option is greater than zero, and the rate of temporary rejection of a specific incoming sender address from a specific host in units of rejections per hour exceeds it, the temporary error is converted into a permanent verification error.

`sender_verify_reject` (Boolean, default = true)
: When this is set, a message is rejected if sender verification fails. If it is not set, a warning message is written to the main and reject logs, and the message is accepted (unless some other error occurs).

Checking Recipient Addresses

There are two approaches that can be taken when handling incoming recipient addresses in SMTP messages. Either the MTA can accept every address and check its deliverability later, or some checking can be done before responding to the `RCPT` command. Exim always checks for unwanted relaying when it receives an `RCPT` command; this is described in the section "Relay Control," later in this chapter. Checking for deliverability can be configured to happen at SMTP time, or to be left until the message is being delivered.

Leaving checks on (nonrelay) addresses until later has some advantages:

- If the checks take some time (for example, when a database lookup is used), avoiding doing them during the SMTP dialog speeds up the SMTP transaction.
- When an address in a local domain turns out to be undeliverable, you may be able to arrange for an informational message to be automatically sent back. (For example, "We have three Charlie Browns at this site: please insert a middle initial.") This can help to reduce the postmaster's workload.

However, there is one big disadvantage: if the address turns out not to be deliverable, and the sender address (despite having been verified) also turns out not to be deliverable, the bounce message is stuck on your system, and is frozen for human attention. On a busy system, large numbers of such stuck messages are a real nuisance. This is a particular problem for universities and other institutions that have a high turnover of people. A lot of mail arrives for cancelled accounts, often from out-of-date mailing lists. Genuine mailing lists usually have a valid sender address to which a bounce can be sent, but an increasing amount of spam mail is being sent using an invalid local part at a valid domain, so that sender verification checks do not catch it unless an expensive callback is used.

Verifying Recipient Addresses

By default, Exim does no checking of recipient addresses during an SMTP transaction, other than ensuring that they conform to the syntax required by RFC 821. However, if:

```
receiver_verify
```

is set, it verifies addresses before responding to `RCPT` commands. This is done exactly as for sender addresses, by running the directors and routers in verify mode. If an address cannot be handled, the response to the `RCPT` command is similar to this:

```
550 Unknown local part jill in <jill@xyz.example>
```

If verification cannot be completed (for example, a database is down), a temporary error code is given, unless the following:

```
receiver_try_verify
```

is set, in which case such an address is accepted, and the incident is logged.

Conditional Recipient Verification

Verification of recipient addresses can be made conditional in a number of ways. First, it can be restricted to certain hosts, or certain hosts can be exempted by setting `receiver_verify_hosts`. For example:

```
receiver_verify_hosts = ! *.cluster.example
```

suppresses recipient verification from the set of hosts whose names end in *cluster.example*.

Second, verification can be restricted to certain addresses by setting `receiver_verify_addresses`. This is normally used to exempt certain domains, or to constrain verification to certain domains. For example, a host that is acting as a relay for a restricted list of remote domains can assume those domains exist without verifying each address,* so it makes sense to restrict verification to its own local domain by a setting such as:

```
receiver_verify_addresses = localdomain.example
```

* An MX backup is a good example of such a host, provided it is not also supporting general outgoing relaying.

Third, verification can be restricted to those messages whose senders match an address list, by setting `receiver_verify_senders`. For example, suppose you have implemented a mechanism for issuing helpful bounce messages for unknown addresses. In order for this to work, these addresses must be accepted at RCPT time (and failed later), but because of the problems of undeliverable bounces, you want to restrict this feature to messages from other domains in your company. By setting:

```
receiver_verify_senders = ! *.mycompany.example
```

you achieve exactly that. Recipients are verified at SMTP time only for messages whose senders are in some other domain.

Testing Recipient Verification

You can test how Exim would respond to a request to verify a particular recipient address by using the *-bv* option:

```
exim -bv minos@crete.example
```

There is a difference between *-bvs* and *-bv* only if your configuration distinguishes between senders and recipients when verifying, for example, by the use of `receiver_verify` on some director or router.

If the configuration of your directors and routers includes any tests of values associated with a sending host (for example, checking the contents of `$sender_host_address`), you can supply values for the test using one of the command-line options whose names start with *-oM* (see the section "Remote Host Information" in Chapter 20).

Explicitly Rejecting Recipients

There are no special options for rejecting specific recipients, as there are for senders. A recipient is rejected if it fails to verify, so if there are certain recipients you do not want to accept, you must configure the routers and directors so that their verification fails. For example, suppose that you allow users on the local host to mail to *root*, but you do not want to accept mail for *root* from outside. The delivery directors must be set up to handle the local part *root* (for example, via an alias), and therefore verification will succeed unless you do something extra. A first director such as this could be used:

```
no_verify_root:
  driver = smartuser
  local_parts = root
  verify_only
  verify_recipient
  fail_verify
```

Because `verify_only` and `verify_recipient` are set, this director is run only when verifying recipients, and the setting of `local_parts` constrains it to just a single local part. As it is a **smartuser** director, it always succeeds, but `fail_verify` converts this success into a verification failure. This means that any attempt to mail to *root* from outside is rejected, but because verification on reception does not apply to locally generated messages, local users are still able to mail to *root*.

Summary of Recipient Rejection Options

The options that control receiver checking are summarized in this section. This checking applies only to messages that arrive over TCP/IP connections.

`receiver_try_verify` (Boolean, default = false)
: If this option is true, envelope recipient addresses on incoming SMTP messages are checked to ensure that they are valid, but if the verification cannot be completed immediately, the message is accepted.

`receiver_verify` (Boolean, default = false)
: If this option is true, envelope recipient addresses on incoming SMTP messages are checked to ensure that they are valid, but if the verification cannot be completed immediately, a temporary error code is given for the `RCPT` command.

`receiver_verify_addresses` (address list, default = unset)
: If set, this option restricts receiver verification to those addresses it matches. The option is inspected only if `receiver_verify` or `receiver_try_verify` is set.

`receiver_verify_hosts` (host list, default = *)
: If `receiver_verify` or `receiver_try_verify` is true, this option specifies a list of hosts to which recipient verification applies.

`receiver_verify_senders` (address list, default = unset)
: This option, if set, allows receiver verification to be conditional upon the sender. It is inspected only if `receiver_verify` or `receiver_try_verify` is set. If the null sender is required in the list of addresses, it must not be the last item, as a null last item in a list is ignored. It is best placed at the start of the list. For example, to restrict receiver verification to messages with null senders and senders in the *.com* and *.org* domains, you could have:

```
receiver_verify
receiver_verify_senders = :*.com:*.org
```

If the null sender is the only entry required, the list should consist of a single colon.

Checking Header Line Syntax

Large numbers of messages with syntactically invalid header lines are now being sent over the Internet. Certain well-known pieces of software seem prone to this, producing illegal lines such as:

```
To: user@domain <user@domain>
To: <mailto:user@domain>
```

It is also common in spam mail to see these examples:

```
To: @
To: <>
```

The option `headers_check_syntax` causes Exim to check the syntax of all headers that can contain lists of addresses (that is, *Sender:*, *From:*, *Reply-To:*, *To:*, *Cc:*, and *Bcc:*) on all incoming messages (both local and remote). This is a syntax check only. No verification is done. If a syntactically invalid header line is found, the message is rejected, unless the following:

```
no_headers_checks_fail
```

is set, in which case the message is accepted, but a warning is written to the reject log. For SMTP messages, as for `headers_sender_verify`, the rejection happens after the end of the data. For non-SMTP messages, a message is written to the standard error stream, and Exim exits with a nonzero return code.

Relay Control

The default Exim configuration permits no relaying of messages. It assumes the simplest kind of environment, where all mail that is accepted from other hosts is addressed to a single local domain that is the same as the host's name. The host is "at the end of the line" for mail delivery. This does not mean that all incoming mail from outside has to be locally delivered; there may be aliases or users' *.forward* files that cause messages to be delivered to other hosts. Such *redirection* is not classed as relaying.

Relaying occurs when a message that is received from another host is passed on to a third host without any reference to a local domain in the recipient address. If a message is sent from *alpha.example* to *beta.example*, with recipient address *homer@beta.example*, no relaying is involved, even if the user *homer* on *beta* has a *.forward* file that sends the message on to another host. This is because the original recipient address is in a local domain.

However, if the recipient is *odysseus@gamma.example*, then we have a case of relaying by *beta*:

```
alpha -> beta -> gamma
```

A single message may have many recipients; some may require relaying whereas others may not. Checking for relay permission must therefore happen for each recipient address independently.

Incoming and Outgoing Relaying

From Exim's point of view, there are two kinds of message relaying, as illustrated in Figure 13-1.

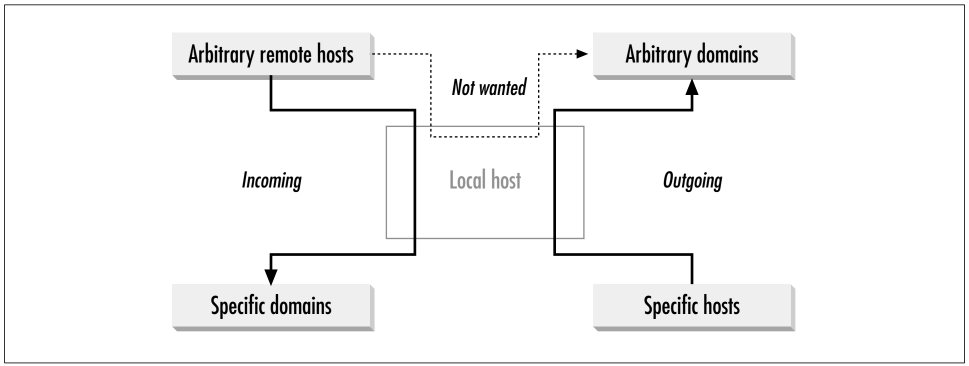


Figure 13-1. Message relaying

A host acting as a gateway or as an MX backup relays messages from arbitrary hosts to a specific set of domains. This is called *incoming relaying*. However, a host acting as a smart host for a number of clients relays messages from those clients to the Internet at large. This is called *outgoing relaying*. What is not wanted is the transmission of mail from arbitrary remote hosts through your system to arbitrary domains.

The same host may fulfill both the incoming and outgoing relay functions, as shown in the figure, but in principle these two kinds of relaying are entirely independent and are controlled by separate options. Large installations often use different hosts for handling incoming and outgoing relaying.

Relay Checking

Checks for unwanted relaying are made on the domains of recipient addresses in messages received from other hosts. None of the relay checking applies when mail is passed to Exim locally using the *-bm*, *-bs*, or *-bS* options, but it does apply when *-bs* is used from *inetd*. The checks are done at the time of the `RCPT` command in the SMTP dialog. The first check is whether the address causes relaying at all; if its domain matches something in `local_domains`, it is handled on the local host as a local address, relaying is not involved, and none of what follows is relevant.

Local parts containing % or @

Addresses such as *"x@y"@z*, where *z* is a local domain, are sometimes used in an attempt to bypass relay restrictions.* Exim treats such addresses as having a local part *x@y*; it does *not* strip off the local domain and treat *x@y* as an entirely new address. Assuming that *x@y* is not a valid local part, this means that the address is rejected, either at SMTP time if `receiver_verify` is set, or later when Exim tries to deliver to it.

Addresses of the form *"x%y"@z* are treated in the same way, unless the "percent hack" has been enabled by setting `percent_hack_domains`. When it is enabled, a new address is constructed from the local part by changing the `%` to an `@`. This is treated as an incoming address, and its domain is retested to ensure that it complies with any relaying restrictions.

Incomplete domains

Exim does not attempt to fully qualify partial domains at `RCPT` time. If an incoming message contains a domain that is not fully qualified, it is treated as a nonlocal, nonrelay domain (unless partial domains are included in `local_domains` or `relay_domains`, but this is not recommended). The use of domains that are not fully qualified is nonstandard, but it is a commonly encountered usage when an MTA is being used as a smart host by some remote MUA. In this situation, however, it would be usual to permit the MUA host to relay to any domain, so in practice there is not normally a problem.

Incoming Relaying

Incoming relaying is controlled by specifying the domains to which an arbitrary host may send via the local host in `relay_domains`. For example, if *alpha.example* is an MX backup host for *beta.example* and *gamma.example*, its configuration would contain:

```
relay_domains = beta.example : gamma.example
```

As another example, if the FooBar Company has a firewall machine through which all mail from external hosts must pass, and this machine's configuration contains:

```
local_domains = foobar.example.com
relay_domains = *.foobar.example.com
```

* Presumably some MTA had this loophole at some time.

then mail from external hosts is rejected, unless it is for the domain *foobar.example.com* itself (which is handled as a local domain) or for another domain that ends in *.foobar.example.com* (for which relaying is permitted).

During the SMTP dialog, when an incoming recipient address has a domain that is not local, it is checked against `relay_domains`. If it matches, the address is accepted; otherwise there is a check for permitted outgoing relaying, as described in the next section.

Automatic relaying for MX backups

If a host is acting as an MX backup for a large number of domains that change frequently, maintaining a list of them for Exim to consult, in addition to the related MX records in the DNS, is a duplication of effort. There is a further option, `relay_domains_include_local_mx`, which, if set, permits relaying for any domain that has an MX record pointing to the local host, whether or not it appears in `relay_domains`.

Turning on `relay_domains_include_local_mx` opens your server to the possibility of abuse in that anyone with access to a DNS zone can list your server in a secondary MX record as a backup for their domain, without your permission. This is not a huge exposure because first, it requires the cooperation of a hostmaster to set up, and second, as their mail is passing through your server, they run the risk of your noticing and (for example) throwing it all away. Nevertheless, the insecurity is there. A safer way of avoiding the maintenance of two different sets of data is to generate both the DNS zone data and Exim's relaying data from a single source.

Outgoing Relaying

If a recipient address is neither for a local domain nor an incoming relay domain, it must be an outgoing relay, and it is accepted only if the sending host is permitted to relay to arbitrary domains. The set of such hosts is defined by `host_accept_relay`. For example, if the FooBar Company's IP network is 192.168.213.0/24, and all hosts on that network send their outgoing mail via the firewall machine, its configuration should contain:

```
host_accept_relay = 192.168.213.0/24
```

This allows the internal hosts, but no others, to use it as a relay to arbitrary domains.

Exim does not make an automatic exception for the loopback IP address, so if you want to permit relaying from processes on the local host that send mail to the loopback address, you need to include 127.0.0.1 (or ::1 on an IPv6 host) in the relay list. For example:

```
hosts_accept_relay = 127.0.0.1 : 192.168.213.0/24
```

Some user agents, notably MH and NMH, send mail by connecting to the loopback address on the local host.

Relaying from authenticated hosts

The option `host_auth_accept_relay` is similar to `host_accept_relay`, except that any client host matching one of its items is permitted to relay only if it has successfully authenticated. This is independent of whether or not it matches `auth_hosts`. See Chapter 15 for details of SMTP authentication.

Relaying using encryption

When Exim is compiled to support SMTP encryption,[*] then you can set `tls_host_accept_relay`. This works in the same way as `host_accept_relay`, except that it insists that an encrypted SMTP session be used for any relaying.

Relaying from specific senders

In addition to the tests on the identity of the host itself, it is possible to restrict outgoing relaying to specific envelope sender addresses. If `sender_address_relay` is set and the host matches `sender_address_relay_hosts` (which defaults to `*`), the sender's address from the `MAIL` command must match one of the patterns in `sender_address_relay` before Exim allows outgoing relaying to an arbitrary domain. For example, a company's mail hub that is relaying mail from clients to the Internet at large can restrict the senders to using the company's own domain:

```
host_accept_relay = 192.168.121.0/24
sender_address_relay = *@plc.co.example
```

If the clients are running their own MTAs that can receive incoming mail, they may from time to time generate bounce messages. If that is the case, you also need to permit empty senders when relaying by changing the `sender_address_relay` setting to:

```
sender_address_relay = : *@plc.co.example
```

Permitting relaying by host or sender

Normally, both the host and the sender must be acceptable before an outgoing relay is allowed to proceed. However, if `relay_match_host_or_sender` is set, an address is accepted for outgoing relaying if *either* the host *or* the sender is

[*] See the section "Encrypted SMTP Connections," in Chapter 15.

acceptable. Setting this option is discouraged because of the ease with which sender addresses can be forged. It was formerly used to permit roaming clients to relay through their "home base" from arbitrary IP addresses, but nowadays SMTP authentication is a safer way of providing that facility.

Summary of Relay Control Options

The options for relay control are summarized in this section. This checking applies only to messages that arrive over TCP/IP connections.

`host_accept_relay` (host list, default = unset)
: This option defines the set of hosts that are permitted to relay via the local host to any arbitrary domain.

`host_auth_accept_relay` (host list, default = unset)
: This option defines a set of hosts that are permitted to relay via the local host to any arbitrary domain, provided the calling host has authenticated itself.

`relay_domains` (domain list, default = unset)
: This option lists domains for which the local host is prepared to act as an incoming relay. Mail for these domains is accepted from any host.

`relay_domains_include_local_mx` (Boolean, default = false)
: This option permits any host to relay to any domain that has an MX record pointing at the local host. It causes any domain with an MX record pointing at the local host to be treated as if it were in `relay_domains`.

`relay_match_host_or_sender` (Boolean, default = false)
: By default, if outgoing relaying controls are specified on both the remote host and the sender address, a message is accepted only if both conditions are met. If `relay_match_host_or_sender` is set, either condition is good enough. Setting this option is discouraged.

`sender_address_relay` (address list, default = unset)
: This option specifies a set of address patterns, one of which the sender of a message must match in order for the message to be accepted for relaying. If it is not set, all sender addresses are permitted.

`sender_address_relay_hosts` (host list, default = *)
: The `sender_address_relay` check is applied only when the sending host matches an item in this list.

Customizing Prohibition Messages

It is possible to add a site-specific message to the error response that is sent when an incoming SMTP command fails for policy reasons; for example, if the sending host is in a host reject list or if relaying is prohibited. This is done by setting the option `prohibition_message`, which causes one or more additional response lines with the same error code and a multiline marker to be output before the standard response line. For example, setting:

```
prohibition_message = contact postmaster@my.site for details
```

causes the response to a `RCPT` command for a forbidden relay to be of the following form:

```
550-contact postmaster@my.site for details
550 relaying to <minos@knossos.example> prohibited by administrator
```

The string is expanded, and so it can perform file lookups if necessary. If it ends up as an empty string, no additional response is transmitted. To make it possible to distinguish between the several different types of administrative rejection, the variable $prohibition_reason is set to a characteristic text string in each case. The possibilities are shown in Table 13-2.

Table 13-2. Prohibition Reasons

Value	Meaning
`host_accept_relay`	The host is not in an `accept_relay` list.
`host_reject`	The host is in a reject list.
`host_reject_recipients`	The host is in a `reject_recipients` list.
`rbl_reject`	The host is rejected by an RBL domain.
`receiver_verify`	Receiver verification failed.
`sender_relay`	The sender is not in a sender relay list.
`sender_reject`	The sender is in a reject list.
`sender_reject_recipients`	The sender is in a `reject_recipients` list.
`sender_verify`	Sender verification failed.

For example, if the configuration contains:

```
prohibition_message = \
  ${lookup{$prohibition_reason}lsearch{/etc/exim/reject.messages}{$value}}
```

and the file */etc/exim/reject.messages* contains (*inter alia*):

```
host_accept_relay:  host not in relay list
```

then a response to a relay attempt might be:

```
550-host not in relay list
550 relaying to <santa@northpole.example.com> prohibited by administrator
```

Because some administrators may want to put in quite long messages, and it isn't possible to get newlines into the text returned from an `lsearch` lookup, Exim treats the vertical bar character as a line separator in this text. If you want the looked-up text to be reexpanded, you can use the `expand` operator. For example, the setting:

```
prohibition_message = \
  ${lookup{$prohibition_reason}lsearch\
  {/etc/exim/reject.messages}{${expand:$value}}}
```

when used with a file entry of the form:

```
host_accept_relay:  Host $sender_fullhost is not permitted to
                    relay |through $primary_hostname.
```

might produce the following:

```
550-Host that.host.name [192.168.3.4] is not permitted to relay
550-through this.host.name.
550 relaying to <penguins@southpole.example.com> prohibited by administrator
```

When the prohibition is due to an entry in a realtime blackhole list (RBL), the variable $rbl_domain contains the RBL domain that caused the prohibition. Some RBL domains use TXT records to provide a message to match each host block. If there is such a record, its text is available in the $rbl_text variable.

Incoming Message Processing

Exim performs various transformations on the original sender and recipient addresses of all messages that it handles, as well as on the messages' header lines. Some of these changes are optional and configurable, while others always take place. All of this processing happens at the time a message is received, before it is first written to the spool. Address rewriting is covered in Chapter 14; the other changes are described here.

RFC 822 makes a provision for header lines starting with the string `Resent-` (for example, *Resent-From:*). It states that in general, these header lines should be treated as containing a set of information that is independent of the set of original fields, and that information for one set should not automatically be taken from the other. If Exim finds any `Resent-` headers in the message, it applies the header transformations, described later in this chapter, only to the `Resent-` header set, thus leaving the others alone.

The UUCP "From" Line

Messages that have come from UUCP (and some other applications) often begin with a line containing the envelope sender and a timestamp, following the word `From` and a space.[*] In the section "Local Sender Addresses," earlier in this chapter, we discuss how this line could be used by a trusted user to supply an envelope sender address for a locally generated message.

For incoming SMTP messages, a UUCP `From` line is not normally recognized. It is syntactically invalid as a header line, so it is treated as the first line of the message's body. However, because there are broken programs that send out SMTP messages with leading `From` lines, there are options to make Exim recognize them in SMTP input. Their contents, however, are always ignored and removed from the message.

For incoming SMTP over TCP/IP, `ignore_fromline_hosts` can be set to a list of hosts for which a `From` line is ignored; for SMTP over the standard input and output (the *-bs* option), `ignore_fromline_local` must be set. These options should be used only as a last resort when broken sending software must be used and cannot be fixed.

In all cases, only one `From` line is recognized. If there is more than one, the second is treated as a data line that starts the body of the message.

The From: Header Line

If an incoming message does not contain a *From:* header, Exim adds one containing the sender's address. This is obtained from the message's envelope in the case of remote messages; for locally generated messages, the calling user's login name and full name are used to construct an address, using the format of this example:

```
From: Zaphod Beeblebrox <zaphod@end.univ.example>
```

The user's full name is obtained from the *-F* command-line option, if set; otherwise, it is obtained by looking up the calling user and extracting the "gecos" field from the password entry. If the "gecos" field contains an ampersand character, this is replaced by the login name with the first letter converted to uppercase, as is conventional in a number of operating systems.

In some environments, the "gecos" field is used to hold more than just the user's name; it might also contain a departmental affiliation or an office or telephone number. The `gecos_pattern` and `gecos_name` options make it possible to extract just the username in such cases. When they are set, `gecos_pattern` is treated as a

[*] A similar line is also used as a separator in "Berkeley format" mailbox files. Do not confuse this with the *From:* header line.

regular expression that is to be applied to the field, and if it matches, `gecos_name` is expanded and used as the user's name.* Numeric variables such as `$1`, `$2`, and so on can be used in the expansion to pick up subfields that were matched by the pattern. In HP-UX, where comma separators are conventionally found, the following can be used:

```
gecos_pattern = ^([^,]*)
gecos_name = $1
```

This extracts everything before the first comma as the user's full name.

In all cases, the username is made to conform to RFC 822 by quoting all or parts of it if necessary. If characters with values greater than 127 appear in a username, Exim encodes it as described in RFC 2047, which defines a way of including non-ASCII characters in header lines. However, if `print_topbitchars` is set, these characters are treated as normal printing characters.

For compatibility with Sendmail, if an incoming non-SMTP message has a *From:* header containing just the unqualified login name of the calling user, this is replaced by an address containing the user's login name and full name, as just described.

The Sender: Header Line

The *Sender:* header line is supposed to contain the address of the originator of a message when this is different to the contents of the *From:* header line. On multiuser systems, it is helpful to record the true identity of the person who sent a message. If one out of several thousand users sends a message containing:

```
From: god@heaven.com
```

and this causes offense, the local postmaster may be grateful for the *Sender:* header when seeking the culprit. However, *Sender:* is of little relevance for messages that originate on single-user systems.

The default behavior of Exim is appropriate for multiuser systems, which is probably what the author of RFC 822 had in mind in the 1980s. *Sender:* header lines are left untouched in messages that arrive over TCP/IP. For other messages, however, unless they are sent by a trusted user using the *-f* or *-bs* command-line option, any existing *Sender:* header lines are removed.

For nontrusted callers, a check is made to see if the address given in the *From:* header is the correct (local) sender of the message. If not, a *Sender:* header giving

* Before matching, any ampersand in the "gecos" field is replaced by the login name, as previously described.

the true sender address is added to the message. This can, however, be disabled by setting the following:

```
local_from_check = false
```

but the envelope sender is still forced to be the login ID at the qualify domain for locally submitted messages.

The sender address that is expected in *From:* is the login ID, qualified with the contents of `qualify_domain`. Some installations may permit the use of prefixes or suffixes to local parts. For example, the addresses:

```
user@example.com
home-user@example.com
work-user@example.com
```

may all refer to the same user, who can make use of the prefix in a filter file to do some automatic mail management. If you do not want the appearance of a prefix in a *From:* to trigger the addition of a *Sender:* header, you can set `local_from_prefix`. For example:

```
local_from_prefix = *-
```

permits any prefix that ends in a hyphen, so that a message containing:

```
From: anything-user@example.com
```

does not cause a *Sender:* header to be added if *user@example.com* matches the actual sender address that is constructed from the login name and qualify domain. The option `local_from_suffix` provides the same facility for suffixes.

The Bcc:, Cc:, and To: Header Lines

If Exim is called with the *-t* option to take recipient addresses from the header lines of a locally submitted message, it removes any *Bcc:* header line that may exist (after extracting its addresses), unless the message has no *To:* or *Cc:* header lines, in which case a *Bcc:* header line with no addresses is left in the message so that it conforms to RFC 822. *Bcc:* header lines are removed from incoming messages only when the *-t* option is used.

If Exim is called to receive a message with the recipient addresses given on the command line and there is no *Bcc:*, *To:*, or *Cc:* header line in the message, it normally adds a *To:* header line and lists the recipients. Some mailing-list software is known to submit messages in this way, and in this case, the creation of a *To:* header line is not what is wanted. If the `always_bcc` option is set, Exim adds an empty *Bcc:* header line instead.

The Return-path:, Envelope-to:, and Delivery-date: Header Lines

A *Return-path:* header line is defined in the RFCs as something the MTA may insert when it does the final delivery of a message, in order to record the envelope sender address.

An *Envelope-to:* header line is not part of the standard RFC 822 header set, but Exim can be configured to add one to the final delivery of a message, in order to record the envelope recipient address.

A *Delivery-date:* header line is not part of the standard RFC 822 header set, but Exim can be configured to add one to the final delivery of a message, to record the time it was delivered.

None of these three header lines should be present in messages in transit, and Exim normally removes any that it finds. This action can be disabled by specifying any or all of the following:

```
return_path_remove = false
envelope_to_remove = false
delivery_date_remove = false
```

The Date: Header Line

If a message has no *Date:* header line, Exim adds one, giving the current date and time.

The Message-id: Header Line

If an incoming message does not contain a *Message-id:* header line, Exim constructs one and adds it to the message. The ID is constructed from Exim's internal message ID, preceded by the letter `E` to ensure that it starts with a letter, and followed by `@` and the primary hostname. Additional information can be included in this header by setting the `message_id_header_text` option.

The Received: Header Line

A *Received:* header is added at the start of every message. The contents of this header are defined by the `received_header_text` option, and Exim automatically adds a semicolon and a timestamp to the configured string. The default setting of this option is:

```
received_header_text = "Received: \
    ${if def:sender_rcvhost {from ${sender_rcvhost}\n\t}\
    {${if def:sender_ident {from ${sender_ident} }}\
    ${if def:sender_helo_name {(helo=${sender_helo_name})\n\t}}}}\
    by ${primary_hostname} \
    ${if def:received_protocol {with ${received_protocol}}} \
    ${if def:tls_cipher {($tls_cipher)\n\t}}\
    (Exim ${version_number} #${compile_number})\n\t\
    id ${message_id} \
    ${if def:received_for {\n\tfor $received_for}}"
```

The reference to $tls_cipher is omitted when Exim is not compiled to support TLS encryption. Note that a string such as this must be enclosed in double quotes so that the escape sequences, such as `\n` and `\t`, are interpreted. The use of conditional expansions ensures that this setting works for both locally generated messages and messages received from remote hosts, thereby giving header lines such as the following:

```
    Received: from scrooge.example ([192.168.12.25] ident=root)
            by marley.example with smtp (Exim 3.22 #1)
            id E0tS3Ga-0005C5-00
            for cratchit@dickens.example; Mon, 25 Dec 2000 14:43:44 +0000
    Received: from ebenezer by scrooge.example with local (Exim 3.22 #2)
            id E0tS3GW-0005C2-00; Mon, 25 Dec 2000 14:43:41 +0000
```

Note the automatic addition of the date and time in the required format.

14

Rewriting Addresses

There are a number of circumstances in which addresses in messages are altered as they are handled by Exim. This can apply both to the messages' envelopes and to their headers. The header lines that may be affected are *Bcc:*, *Cc:*, *From:*, *Reply-To:*, *Sender:*, and *To:*. Some of these changes happen automatically, whereas others are explicitly configured by the administrator.

Automatic Rewriting

One case of automatic rewriting is the addition of a domain to an unqualified address, as discussed in Chapter 13, *Message Reception and Policy Controls*. This qualification is applied to addresses in header lines as well as to those in envelopes. For example, if a message is sent on a host where `qualify_domain` is set to *crete.example* by this command:

```
$ exim daedalus
To: daedalus
...
```

the unqualified local part *daedalus* is transformed into the fully qualified address *daedalus@crete.example*, both in the envelope and in the *To:* header line. Messages that arrive from other hosts should not contain unqualified addresses; you need to set `sender_unqualified_hosts` and/or `receiver_unqualified_hosts` if you want to allow such messages to be accepted (as described in Chapter 13).

The other case in which automatic rewriting happens is when an incomplete domain is given. The routing process may cause this to be expanded into the full domain name within the current encompassing domain. For example, a header such as:

```
To: minos@knossos
```

might be rewritten as:

```
To: minos@knossos.crete.example
```

if encountered on a host within the *crete.example* domain. Strictly, an MTA should not do any deliveries of a message until all its addresses have been routed, in case any of the header lines have to be changed as a result of routing. Otherwise it runs the risk of sending different copies of the message to different recipients.

However, doing this in practice could hold up many deliveries for unreasonable amounts of time in messages when one address could not immediately be routed (because of DNS timeouts, for example). Exim therefore does not delay other deliveries when routing of one or more addresses is deferred. Since it is normally addresses for distant domains that cannot immediately be routed, and such addresses are not normally rewritten by this process, the risk of getting it wrong is minimal.

Configured Rewriting

Some people believe that configured rewriting is a Mortal Sin, because "MTAs should not tamper with messages." Others believe that life is not possible without it. Exim provides the facility; you do not have to use it. In general, rewriting addresses from your own domains has some legitimacy. Rewriting other addresses should be done only with great care and in special circumstances. The author of Exim believes that rewriting should be used sparingly, and mainly for "regularizing" addresses in your own domains. Although rewriting recipient addresses can be used as a routing tool, it is not intended for this purpose, and this use of rewriting is not recommended.

There are two commonly encountered circumstances where address rewriting is used, as illustrated by these examples:

- The company whose domain is *hitch.example* has a number of machines that exchange mail with each other behind a firewall, using the hostnames as mail domains, but only a single gateway to the outer world. The gateway removes the local hostnames from addresses in outgoing messages, so that, for example, *fp42@vogon.hitch.example* becomes *fp42@hitch.example*. A rewriting rule that implements this is:

  ```
  *@*.hitch.example  $1@hitch.example
  ```

- A host rewrites the local parts of its own users to remove login names and replace them by real-world names, so that, for example, *fp42@hitch.example* becomes *Ford.Prefect@hitch.example*. This can be done by a rewriting rule of this form:

```
*@hitch.example   ${lookup{$1}dbm{/etc/realnames}\
                  {$value}fail}@hitch.example frsF
```

We explain shortly how these rewriting rules operate. The two kinds of rewriting are not mutually exclusive, and very often both are done by having both rules in the configuration.

The order in which addresses are rewritten is undefined, except that envelope sender addresses are always rewritten before any header lines are rewritten, so if a rewrite of an address in a header line refers to `$sender_address`, it is the rewritten value that is used. However, you cannot assume that, for example, the *From:* header is always rewritten before the *To:* header.

Configured address rewriting can take place at several different stages of a message's processing. Rewriting happens when a message is received, but it can also happen when a new address is generated during directing or routing (for example, by aliasing), and when a message is transported.

Two different kinds of address rewriting can be set up by an Exim administrator. They are called *general* and *per-transport* rewriting. They operate in a similar way, but at different times. General rewriting applies to all copies of a message, whereas per-transport rewriting applies only to those copies of a message that pass through a particular transport.

General Rewriting

General rewriting is defined by a set of rules that are given in the sixth part of the runtime configuration file. Each rule specifies the types of address on which it operates, and Exim applies the rules to each address when it first encounters it.

- A message's sender address, its original recipient addresses, and the addresses in its header lines are rewritten as soon as the message is received, before the start of any delivery processing. This happens only once. If the message was received from a host that is permitted to send unqualified addresses, they are qualified before rewriting. Other hosts are required to send fully qualified addresses in the envelope for the message to be accepted, but there may still be unqualified addresses in the header lines. Such addresses are left entirely alone; they are neither qualified nor rewritten.

- Recipient addresses that are generated during delivery (for example, by aliasing or forwarding), or by the `new_address` option of the **smartuser** director, are rewritten at the time they are generated, unless `no_rewrite` is set on the relevant director.

Addresses in header lines that are generated during delivery (that is, those that are added by routers, directors, transports, or a system filter), are not subject to general rewriting.

Per-Transport Rewriting

If the corporate gateway that was used as an earlier example is a host that does nothing but relay mail to the outside world, general rewriting rules can be used because rewriting is required for all deliveries.

However, not every site has the luxury of a separate host just to do outgoing mail relaying. If the same host is handling both onsite and offsite deliveries, there is a problem if the requirement is to rewrite addresses only in those copies of messages that are going offsite. The problem arises because Exim keeps only one copy of a message, however many recipients it has. If a message has both local and remote recipients, the requirement for rewriting only for remote delivery cannot be met by general rewriting.

To solve this problem, per-transport rewriting was introduced in Exim Release 3.20.* It allows addresses in header lines to be rewritten at transport time (that is, as the message is being copied to a destination). Unlike general rewriting, envelope addresses cannot be rewritten by this means. You can rewrite the envelope sender by using the `return_path` option on a transport, but you cannot rewrite recipient addresses at transport time.

Per-transport rewriting is configured by setting the `headers_rewrite` option on a transport to a colon-separated list of rewriting rules. Each rule is in exactly the same form as one of the general rewriting rules that are applied when a message is received (see the following section). For example:

```
headers_rewrite = a@b c@d f : \
                  x@y w@z
```

changes *a@b* into *c@d* in `From:` header lines, and *x@y* into *w@z* in all address-bearing header lines. However, only the message's original header lines, as well as any

* If you are running an earlier release, the only way you can solve this problem is to run two different versions of Exim: one that rewrites and one that does not. This is messy to set up and maintain. Upgrading to a later Exim release is a better option.

that were added by a system filter, are rewritten. Note that this is different to general rewriting, which does not apply to header lines added by a system filter. If a router, director, or transport adds header lines, these are not affected.

Rewriting Rules

For both general and per-transport rewriting, the entire set of rules is applied to one address at a time. In other words, Exim does not go through the rules once, applying each one to every relevant address (which is one way it might have worked). Instead, it completely rewrites each address before moving on to the next one.

For each address, the rules are scanned in order of definition, with each one potentially changing the address so that a replacement address from an earlier rule can itself be rewritten as a result of the application of a later rule. However, there are some cases where scanning stops after a particular rule has been applied.

Here is a very simple rewriting rule that just turns one explicit address into another:

```
ph10@workshop.exim.example   P.Hazel@exim.example
```

As a general rewriting rule, this would occupy a line by itself and apply to all instances of the address. As a per-transport rule, it would be the value of `headers_rewrite`:

```
headers_rewrite = ph10@workshop.exim.example  P.Hazel@exim.example
```

In this case, it would apply only to addresses in header lines. Further examples are shown here as general rewriting rules, but they could equally be used as per-transport rules.

In many cases, some kind of wildcard matching is employed in rewriting rules, and lookups can be used to vary the replacement address. The following configuration uses two rules to implement the most common forms of rewriting for the domain *exim.example*:

```
*@*.exim.example  $1@exim.example
*@exim.example    ${lookup{$1}dbm{/etc/realnames}\
                  {$value}fail}@exim.example frsF
```

The first rule removes the first component of domain names that end in *.exim.example*, and the second rule converts the local part by means of a file lookup. Thus, addresses in the *exim.example* domain are rewritten in two stages. The `frsF` that appears at the end of the second rule is a string of flag characters, which are explained in the following section.

Format of Rewriting Rules

In general, each rewriting rule is of the form:

```
pattern  replacement  flags
```

The pattern is terminated by whitespace, and it matches those addresses that are to be rewritten by this rule. The flags are single characters, some of which indicate the address location (header line, envelope field) to which the rule applies; other flags control how the rewriting takes place. Both the pattern and the address location flags must match for a rule to be "triggered." The allowed formats for patterns and flags are described later.

The replacement string is also terminated by whitespace, unless it is enclosed in double quotes. If quotes are used, normal quoting conventions apply inside them. A common configuration error is to forget to quote replacement strings that contain whitespace.

Applying Rewriting Rules

If the replacement string for a rule is a single asterisk, an address that matches is not rewritten by that rule, and no subsequent rewriting rules are scanned for the address. For example:

```
hatta@lookingglass.example  *
```

specifies that *hatta@lookingglass.example* is never to be rewritten. Otherwise, the replacement string is expanded and must either yield a fully qualified address or be terminated by a forced failure in a lookup or conditional expansion item. A forced failure causes the rewriting rule not to modify the address; it is equivalent to generating the following:

```
$local_part@$domain
```

as a replacement (but a bit more efficient). Any other kind of expansion failure (for example, a syntax error) causes the entire rewriting operation to be abandoned, and an entry to be written to the panic log.

Within a replacement string, the numerical variables (`$1`, `$2`, and so on) are set up according to the type of pattern that matched the address, and the variables $local_part and $domain refer to the address that is being rewritten. Any letters in these variables retain their original case; they are not lowercased.

Conditional Rewriting

The behavior of forced expansion failures means that the conditional features of string expansion can be used to implement conditional rewriting. For example, the following would apply a rewriting rule only to messages that originate outside the local host:

```
*@*.hitch.example  "${if !eq {$sender_host_address}{}\
                   {$1@hitch.example}fail}"
```

The value of $sender_host_address is the empty string for locally originated messages; this rule restricts rewriting to cases when it is not empty (that is, to cases where the message came from a remote host). The forced failure causes the rewriting rule to be abandoned for locally originated messages, but subsequent rules are still applied to the address. Note that quotes have to be used for the rule in this example because it contains whitespace characters.

Lookup-Driven Rewriting

Rewriting that is entirely lookup-driven can be implemented by a rule of the form:

```
*@*   ${lookup ...
```

with a lookup key derived from `$local_part` and `$domain`. The pattern matches every address, so the behavior is entirely controlled by the expansion of the replacement string.

Rewriting Patterns

The source pattern in a rewriting rule can be in one of the forms in the following list. It is not enclosed in quotes, and there is no special processing of any characters; it is not expanded. If it is a regular expression, backslash characters do not need to be doubled.

- An address containing a local part and a domain, either of which may start with an asterisk, implying independent wildcard matching, for example:

  ```
  *@orchestra-land.example
  ```

  If the domain is specified as a single @ character, it matches the primary hostname. After matching, the numerical variables refer to the character strings matched by asterisks, with $0 referring to the entire address, $1 referring to the first asterisk, and $2 referring to the second asterisk, if present. For example, if the pattern:

  ```
  *queen@*.example
  ```

  is matched against the address *hearts-queen@wonderland.example*, the three variables would be set as follows:

  ```
  $0 = hearts-queen@wonderland.example
  $1 = hearts-
  $2 = wonderland
  ```

Note that if the local part does not start with an asterisk, but the domain does, it is $1 that contains the wild part of the domain.

- A local part, possibly starting with an asterisk, and a lookup item (as in a domain list), for example:

  ```
  root@lsearch;/etc/special/domains
  ```

  If there is an asterisk in the local part, the value of the wild part is placed in the first numerical variable. If the lookup is a partial one, the wild part of the domain is placed in the next numerical variable, and the fixed part of the domain is placed in the succeeding variable. Thus, for example, if the address *foo@bar.baz.example* is processed by a rewriting rule using the pattern:

  ```
  *@partial-dbm;/some/dbm/file
  ```

  and the key in the file that matches the domain is `*.baz.example`, the three variables would be set as follows:

  ```
  $1 = foo
  $2 = bar
  $3 = baz.example
  ```

  If the address *foo@baz.example* is looked up, this matches the same wildcard file entry, and in this case $2 is set to the empty string but $3 is still set to `baz.example`. If a nonwild key is matched in a partial lookup, $2 is set to the empty string and $3 is set to the whole domain again. For nonpartial lookups, no numerical variables are set.

- A local part, possibly starting with an asterisk and a regular expression (as in a domain list), for example:

  ```
  *.queen@^(wonderland|lookingglass)\.example$
  ```

  If there is an asterisk in the local part, the value of the wild part is placed in the first numerical variable. Any substrings captured by the regular expression are placed in numerical variables starting at $1 if there is no asterisk in the local part, or at $2 if there is.

- A lookup without a local part, for example:

  ```
  partial-dbm;/rewrite/database
  ```

  This works similarly for an address list configuration item; the domain is first looked up, possibly partially, and if that fails, the whole address is looked up (not partially). When a partial lookup succeeds, the numerical variable $1 contains the wild part of the domain, and $2 contains the fixed part. The `@@` form of address list lookup can also be used.

- A single regular expression. This is matched against the entire address, with the domain part lowercased. After matching, the numerical variables refer to the bracketed capturing subexpressions, with $0 referring to the entire address. For example, if the following pattern:

  ```
  ^(red|white)\.king@(wonderland|lookingglass)\.example$
  ```

  is matched against the address *red.king@lookingglass.example*, then the variables would be set as follows:

  ```
  $0 = red.king@lookingglass.example
  $1 = red
  $2 = lookingglass
  ```

  Note that, because the pattern part of a rewriting rule is terminated by whitespace, no literal whitespace may be present in the regular expression.

Rewriting Flags

There are several different kinds of flag that may appear on rewriting rules:

- Flags that specify header lines and envelope fields to which the rule applies: `E, F, T, b, c, f, h, r, s, t`
- Flags that control the rewriting process: `Q, q, R, w`
- A flag that specifies rewriting at SMTP time: `S`

For per-transport rewriting rules, which apply only to header lines, the flags `E`, `F`, `S` and `T` are not permitted.

Flags Specifying What to Rewrite

If none of the flag letters in Table 14-1, nor the `S` flag (see later in this chapter) are present, the rewriting rule applies to all header lines and (for general rewriting but not for per-transport rewriting) to both the sender and recipient fields of the envelope. Otherwise, the rewriting rule is used only when rewriting addresses from the appropriate sources.

Table 14-1. Flags to Select Addresses

Flag	Meaning
`E`	Rewrite all envelope fields.
`F`	Rewrite the envelope From field.
`T`	Rewrite the envelope To field.
`b`	Rewrite the *Bcc:* header.
`c`	Rewrite the *Cc:* header.

Table 14-1. Flags to Select Addresses (continued)

Flag	Meaning
`f`	Rewrite the *From:* header.
`h`	Rewrite all headers.
`r`	Rewrite the *Reply-To:* header.
`s`	Rewrite the *Sender:* header.
`t`	Rewrite the *To:* header.

Thus, in this example, which was given earlier:

```
*@*.exim.example  $1@exim.example
*@exim.example    ${lookup{$1}dbm{/etc/realnames}\
                  {$value}fail}@exim.example frsF
```

the first rule applies to all addresses, but the second one is used only for the *From:*, *Reply-to:*, and *Sender:* header lines and the envelope sender (From) field.

Flags Controlling the Rewriting Process

There are four flags that control the way the rewriting process works. These take effect only when a rule is invoked; that is, when the address is of the correct type (matches the selection flags) and also matches the pattern:

- If the `Q` flag is set on a rule, the rewritten address is permitted to be an unqualified local part. It is qualified with `qualify_recipient`. In the absence of `Q`, the rewritten address must always include a domain. There is not much point in writing a rule such as:

  ```
  aaaa@domain.example   bbbb   Q
  ```

  because you might just as well write the qualifying domain in the replacement address. However, the flag can be useful if the replacement involves a file lookup that just produces local parts.
- If the `q` flag is set on a rule, no further rewriting rules are considered for the current address, even if no rewriting actually takes place because of a forced failure in the expansion. The `q` flag does not apply if the address is of the wrong type (does not match the selection flags) or does not match the pattern.
- The `R` flag causes a successful rewriting rule to be reapplied to the new address, up to 10 times. It can be combined with the `q` flag to stop rewriting once it fails to match (after at least one successful rewrite). The `R` flag is generally of use in gateway environments where different styles of addressing are used. An example is given in connection with the `S` flag later.

- When an address in a header line is rewritten, the rewriting normally applies only to the working part of the address, with any comments and RFC 822 "phrase" left unchanged. For example, the rule:

  ```
  fp42@*.hitch.example  prefect@hitch.example
  ```

  would change the following:

  ```
  From: Ford <fp42@restaurant.hitch.example>
  ```

  into:

  ```
  From: Ford <prefect@hitch.example>
  ```

  Sometimes there is a need to replace the whole RFC 822 address item, and this can be done by adding the flag letter `w` to a rule. If this is set on a rule that causes an address in a header line to be rewritten, the whole address is replaced, not just the working part. For example, the rule:

  ```
  fp42@*.hitch.example  "\"F.J. Prefect\" <prefect@hitch.example>"  w
  ```

  changes the following:

  ```
  From: Ford <fp42@restaurant.hitch.example>
  ```

  into:

  ```
  From: "F.J. Prefect" <prefect@hitch.example>
  ```

  The replacement must be a complete RFC 822 address, including the angle brackets if necessary. When the `w` flag is set on a rule that causes an envelope address to be rewritten, all but the working part of the replacement address is discarded because envelope addresses do not contain "phrase" or comment items.

The SMTP-Time Rewriting Flag

The rewrite flag `S` specifies a rule that applies to incoming envelope addresses at SMTP time, as soon as each `MAIL` or `RCPT` command is received and before any other processing; even before syntax checking. This form of rewrite rule allows for the handling of addresses that are not compliant with RFC 821 (for example, UUCP "bang paths" in SMTP input, or malformed addresses from broken SMTP clients).

Because the input for a rewriting rule with the `S` flag is not required to be a syntactically valid address, the pattern must be a regular expression, and the variables $local_part and $domain are not available during the expansion of the replacement string. The pattern is matched against the entire text supplied by a `MAIL` or `RCPT` command, including any enclosing angle brackets, and the result of rewriting replaces the original address in the `MAIL` or `RCPT` command. For example, suppose one of your local SMTP clients is broken and sends malformed SMTP commands such as this:

```
RCPT TO: internet:ceo@plc.example
```

There are two things wrong with this address: the lack of surrounding angle brackets and the presence of the unwanted string `internet:` at the beginning. Obviously the best thing would be to get the client fixed, but sometimes this is not possible, at least not quickly. Using the `S` rewriting flag you can arrange for Exim to patch up such bad addresses:

```
^\s*internet:(.*)$  <$1>  S
```

The regular expression pattern matches addresses that start with `internet:` (with optional leading whitespace), and arranges to capture the remainder by means of the parentheses. The replacement string wraps the captured substring in the angle brackets that are required in SMTP commands, so the result is treated as if the command was:

```
RCPT TO:<ceo@plc.example>
```

Another case where this kind of rewriting is useful is when interfacing to systems that use UUCP bang-path addressing, in which addresses are of the form:

```
host1!host2!host3...!user
```

Exim supports only Internet domain-based addressing, and so does not recognize bang paths. However, in some cases, rewriting can be used to convert bang-path addresses. If this is done at SMTP time using the `S` flag, the rewritten addresses are subject to the normal verification and relay checking, which is what you want.* The following rules convert bang paths into more conventional Internet addresses at SMTP time:

```
^(?=.*?!)(?!.*?@)(.*)$              $1@bang.path       S
^([^!]+)!([^%]+)([^@]*)@bang\.path$  $2%$1$3@bang.path  SR
^(.*)%([^@]*)@bang\.path$           $1@$2              S
```

The first rule recognizes strings that contain at least one exclamation mark but no `@` characters, and adds the pseudodomain *bang.path*, thereby allowing the other two rules to operate on them. Conventional Internet addresses that happen to contain exclamation marks are not affected by this rewriting.

The second rule changes the local part of these special addresses by reversing the order of the parts and replacing the exclamation marks with percent signs. For example, *a!b@bang.path* becomes *b%a@bang.path*, and *a!b!c!d@bang.path* becomes *d%c%b%a@bang.path*. The rule needs the `R` (repeat) flag, because each

* Since a bang-path address is a syntactically valid local part, you could configure Exim to accept unqualified addresses, and then later rewrite local parts containing exclamation marks. This is not, however, a good idea, because it bypasses relay checking.

time it runs, it handles only one exclamation mark. The final rule removes the pseudodomain, and changes the final percent sign into an `@`.

Using these rules, a two-part bang path such as *a!b* is turned into *b@a*, and a longer path such as *a!b!c!d* becomes *d%c%b@a*. This notation is sometimes called the "percent hack," and has been used in Internet addressing for explicit routing of mail, though it is not a standard. Nowadays it is falling out of use.

A Further Rewriting Example

The ability to rewrite addresses may be used in lots of different ways. The most common use of rewriting is for removing local hostnames, and converting login names to "real names" in messages that are leaving a local network for the wider Internet. Earlier we showed a simple way of implementing such rewriting:

```
*@*.exim.example  $1@exim.example
*@exim.example    ${lookup{$1}dbm{/etc/realnames}\
                  {$value}fail}@exim.example frsF
```

The first rule removes local hostnames, and the second rewrites local parts that it recognizes in header and envelope sender fields, assuming that the local parts are unique among the local hosts.

This example can be extended to provide additional functionality, such as rejecting messages whose local sender addresses cannot be recognized. We show the complete configuration first, and then explain how it works. There are five rules:

```
^(?>.*)(?<!\.exim\.example)  *
root@*.exim.example "admin@exim.example (root@$1)"  hFwq
*@*.exim.example \
  ${lookup{$local_part@$2}lsearch{/etc/realnames}{$value}\
  {"$1@$2-is-not-known"}}@exim.example  Fq
*@*.exim.example \
  ${lookup{$local_part@$2}lsearch{/etc/realnames}{$value@exim.example}\
  {$sender_address}}  fsrq
*@*.exim.example \
  ${lookup{$local_part@$2}lsearch{/etc/realnames}{$value}{unknown}}\
  @exim.example
```

Because we are rewriting only those addresses that end in *.exim.example*, we can save some resources by having an initial rule that recognizes other domains and abandons any attempt to rewrite them:

```
^(?>.*)(?<!\.exim\.example)  *
```

The pattern is a regular expression that matches all addresses that do not end in *.exim.example*, and the single asterisk as a replacement string means "do not rewrite this address and do not scan any more rules." This means that only addresses that end with *.exim.example* are passed to the remaining rules.

Some explanation of the regular expression is needed. There are several ways you can write a pattern to implement "ends with," but this is the most efficient. The start of the expression:

```
^(?>.*)
```

matches the entire string, `^` matches the start of the string, and `.*` matches any number of arbitrary characters, but because it is enclosed in `(?>)` parentheses, no backtracking is permitted, so having reached the end of the string, the current point stays there. However, the pattern itself is not finished. We still have:

```
(?<!\.exim\.example)
```

This is a negative backward assertion; it checks that the characters immediately preceding the current position (that is, those at the end of the string) are `.exim.example`. If it fails, the pattern match fails because no backtracking is permitted to try this test at any other position. If this test succeeds, the pattern match succeeds because the end of the pattern has been reached.

The more "obvious" regular expression is to check that a string ends with `.exim.example`, namely:

```
\.exim\.example$
```

This is less efficient because it scans through the string character by character, looking for a dot; whenever it finds one, it checks to see if it is followed by `exim.example` and the end of the string. This is more work than a single check at the end of the string.

The second rewriting rule in this example handles mail from *root*. Such mail may be generated by *cron* or other system jobs on local hosts. Although it is usually addressed to local users, there is always the possibility that such users have forwarded their mail offsite. If there is more than one local host, rewriting *root* in the same way as a user login name loses information about which local host's *root* actually sent the message. This rewriting rule is one way of preserving this information:

```
root@*.exim.example "admin@exim.example (root@$1)"  hFwq
```

The pattern matches *root* at any of the local hosts, and the `hF` flags restrict this rule to header lines and the envelope sender. The new address is a standard one but retains the original hostname in a comment. The `w` flag ensures that the comment is retained in any header lines that are rewritten, so that, for example:

```
From: Charlie Root <root@host1.exim.example>
```

is rewritten as:

```
From: admin@exim.example (root@host1)
```

This does, of course, allow the local hostname to remain in the message. Sites that are paranoid about hiding their local hostnames would not want to do this.* The `q` flag on this rule causes rewriting to cease after the rule has been obeyed, so the subsequent rules do not apply to *root* addresses.

The third rule rewrites the envelope sender of the message by looking up the address, minus the terminating *.exim.example*, in a file containing lines such as this:

```
jc@host1:    J.Caesar
jc@host2:    Jiminy.Cricket
```

which might be called */etc/realnames*. In other words, it allows for nonunique local parts among the local hosts. Provided this file is not too big, using a linear search is acceptable.† The pattern does not need to test the domain, because we know that the rule is applied only to domains ending in *.exim.example*, so the rule is as follows:

```
*@* ${lookup{$local_part@$2}lsearch{/etc/realnames}{$value}\
  {"$1@$2-is-not-known"}}@exim.example  Fq
```

The `F` flag ensures that this rule is applied only to envelope senders. If the address cannot be found in the file, it is rewritten to a magic sequence. An unknown sender address such as *xxxx@host1.exim.example* is turned into the following:

```
"xxxx@host1-is-not-known"@exim.example
```

by this rule. The idea here is that, provided incoming sender addresses are being verified, this rewritten address will fail to verify, and so the message will not be accepted by the gateway for onward transmission if its sender address is not in the list of local addresses.

The fourth rule rewrites sender addresses within the message's *From:*, *Sender:*, and *Reply-To:* header lines. It is almost the same as the previous rule, except that failure to look up the original address is not treated as such a serious error here; an unknown address is replaced by the sender address from the envelope:

```
*@* ${lookup{$local_part@$2}lsearch{/etc/realnames}{$value@exim.example}\
  {$sender_address}}     fsrq
```

Exim always rewrites the envelope sender address before it rewrites header lines,‡ so we know that the sender address has been validated.

The final rule has no flags, and so in theory applies to all addresses, but because of the use of the `q` flag in previous rules, it is only ever applied to envelope

* They would probably also want to remove the *Received:* header lines that show local hostnames.

† Once it gets above a hundred lines or so, it should be converted into some kind of indexed lookup, for example, a cdb or DBM file (see Chapter 16, *File and Database Lookups*).

‡ This is the only ordering of rewrites that is specified.

recipients and the addresses in recipient header lines *To:* and *Cc:* (and *Bcc:* if present). It replaces unknown local parts in our domain with `unknown`:

```
*@* ${lookup{$local_part@$2}lsearch{/etc/realnames}{$value}{unknown}}\
  @exim.example
```

Testing Rewriting Rules

Exim's general rewriting configuration can be tested by the *-brw* command-line option. This takes an address (which can be a full RFC 822 address) as its argument. The output is a list of how the address would be transformed by the general rewriting rules for each of the different places it might appear; that is, for each different header line and for the envelope sender and recipient fields. For example:

```
exim -brw ph10@workshop.exim.example
```

might produce the output:

```
  sender: Philip.Hazel@exim.example
    from: Philip.Hazel@exim.example
      to: ph10@workshop.exim.example
      cc: ph10@workshop.exim.example
     bcc: ph10@workshop.exim.example
reply-to: Philip.Hazel@exim.example
env-from: Philip.Hazel@exim.example
  env-to: ph10@workshop.exim.example
```

which shows that general rewriting has been set up for that address when used in any of the source fields, but not when it appears as a recipient address. If the `S` flag is set for any rewriting rules, another line is added to the output, showing the rewriting that would occur at SMTP time.

15

Authentication, Encryption, and Other SMTP Processing

We mention SMTP authentication and encryption in earlier chapters, and also mention several aspects of the other processing that happens when Exim sends or receives messages using SMTP. In this chapter, we describe how SMTP authentication and encryption works, and how you can configure Exim to make use of them. After that we go into some detail about general SMTP processing, for those that want to know more about the nitty-gritty.

SMTP Authentication

The original SMTP protocol, designed for a small, cooperative network consisting mostly of fairly large, multiuser hosts, had no concept of authentication. All hosts were equal, and any host could send mail to any other for onward delivery as best it could. Today's Internet is very different. The concept of *servers* and *clients* has arisen, and hosts that do mail relaying are servers that are configured to allow it to happen only when the mail arrives from an approved client.

One way of controlling relaying is by checking the sending host (as discussed in the section "Relay Control," in Chapter 13, *Message Reception and Policy Controls*). For example, you might permit relaying only from clients on your local network, using a configuration such as:

```
host_accept_relay = 192.168.5.224/27
```

but that approach does not work in cases such as the following:

- An employee with a laptop is away from base, and wants to be able to connect from arbitrary locations and send outgoing mail via the server back at home. Even without a laptop, someone might want to do this from a cybercafe, or other "foreign" client.

- An employee has a dial-up ISP account at home that uses a different IP address each time a new connection is made, so `host_accept_relay` cannot be used.
- The local network is not a strong-enough restriction; only those persons who are authorized may send mail via the server from a workstation.

SMTP authentication (RFC 2554) was invented as one way of solving these problems. It works like this:

- When a server that supports authentication is sent an `EHLO` command, it advertises a number of authentication *mechanisms*. For example, the response to `EHLO` might be:

  ```
  250-server.test.example Hello client.test.example [10.0.0.1]
  250-SIZE
  250-PIPELINING
  250-AUTH LOGIN CRAM-MD5
  250 HELP
  ```

  The second-to-last line advertises the `LOGIN` and `CRAM-MD5` authentication mechanisms.
- When a client wants to authenticate, it sends the SMTP command `AUTH`, followed by the name of an authentication mechanism, for example:

  ```
  AUTH LOGIN
  ```

  The command may optionally contain additional data.
- The server replies with a *challenge* string associated with a response code beginning with the digit 3, indicating "more data needed." The challenge string may be a simple prompt such as "Please enter a password."
- The client answers the challenge by sending a response string.
- The server may send another challenge, and the challenge-response sequence can be repeated any number of times, including zero. If, for example, all the authentication data is sent in the `AUTH` command, no more may be needed, so there are no challenges. All the data is encoded in base-64 so that it can include all 256 byte values. Note that this is not encryption. Anybody who intercepts the transmission is able to decode it into its original form.
- Eventually, the server responds with a return code indicating success or failure of the authentication attempt, or a temporary error code if authentication could not be completed.

Once a client has authenticated, the server may permit it to do things that unauthenticated clients are not allowed to do. What these are is entirely up to the management of the server.

Authentication Mechanisms

Several different authentication mechanisms have been published. Exim supports three of them: PLAIN, LOGIN, and CRAM-MD5, which are used by various popular user agents that submit mail to a server using SMTP. However, since not everybody is interested in SMTP authentication, the code is not included in the Exim binary unless the build-time configuration explicitly requests it.

Before describing how Exim is configured to support SMTP authentication, we need to explain how the three common authentication mechanisms work.

PLAIN Authentication

PLAIN authentication is described in RFC 2595. It requires that three data strings be sent with the `AUTH` command, separated by binary zero characters. The second and third strings are a user/password pair that can be checked by the server. The first string is not relevant to SMTP authentication, and is normally empty.* No additional challenge strings are sent. Here is an example of an authentication exchange, where the lines sent by the client and the server are identified by C and S, respectively:

```
C: AUTH PLAIN AHBoMTAAc2VjcmV0
S: 235 Authentication successful
```

The base-64 string `AHBoMTAAc2VjcmV0` is an encoding of:

```
<nul>ph10<nul>secret
```

where `<nul>` represents a binary-zero byte. The first data string is empty, the username is `ph10`, and the password is `secret`.

PLAIN authentication is efficient in that it requires only a single command and response. The password must be held in clear on the client host, but can be kept encrypted on the server, exactly as it is for login passwords. However, unless an encrypted SMTP connection is used, the data travels over the network unencrypted, and is vulnerable to eavesdropping.

LOGIN Authentication

LOGIN authentication is not described in any RFC, but it is used by the user agent Pine. Like PLAIN authentication, it is based on a user/password combination, but

* The mechanism is designed for use in protocols other than SMTP, where a specific user identity is used for subsequent operations (for example, to run a login session); the first string can specify a different user from the one whose password was checked.

each of these is prompted for separately, so an authentication exchange might look like this:

```
C: AUTH LOGIN
S: 334 VXNlciBOYW1l
C: cGgxMA==
S: 334 UGFzc3dvcmQ=
C: c2VjcmV0
S: 235 Authentication successful
```

Unencoded, this is:

```
C: AUTH LOGIN
S: 334 User Name
C: ph10
S: 334 Password
C: secret
S: 235 Authentication successful
```

LOGIN authentication is less efficient than PLAIN, because three interactions are required. Like PLAIN authentication, the username and password are transmitted in clear. Some people have argued that it is "safer" because the username and password do not travel in the same packet, though this does not seem to be a very strong argument.

CRAM-MD5 Authentication

CRAM-MD5 authentication (RFC 2195) avoids transmitting unencrypted passwords over the network. The server sends a single challenge string, and the client sends back a username, followed by a space and the MD5 digest* of the challenge string concatenated with a password. The server computes the MD5 digest of the same string and compares this with what it has received. For example:

```
C: AUTH CRAM-MD5
S: 334 PDE4OTYuNjk3MTcwOTUyQHBvc3RvZmZpY2UucmVzdG9uLmljaS5uZXQ+
C: dGltIGRkOTJiNGJiMzRhZmFhNzBmMjkwNWVkZDMxOTZhNTU3
S: 235 Authentication successful
```

Unencoded, this is:

```
C: AUTH CRAM-MD5
S: 334 <1896.697170952@postoffice.reston.example>
C: tim dd92b4bb34afaa70f2905edd3196a557
S: 235 Authentication successful
```

The string `dd92b4bb34afaa70f2905edd3196a557` is the MD5 digest of:

```
<1896.697170952@postoffice.reston.example>secret
```

* An MD5 digest is a 16-byte cryptographic hash computed from an arbitrary text string in such a way as to minimize the chances of two strings having the same digest. See RFC 1321.

but that string cannot be recovered from the digest by someone that intercepts it, and the digest cannot be reused, because the challenge string is different each time.

CRAM-MD5 requires only two interactions, and avoids transmitting the password in clear, but the disadvantage is that the password must be held in clear on the server as well as on the client.

Choice of Authentication Mechanism

If you are setting up an Exim client to use a remote server, and you do not know which authentication mechanisms it supports, you can use Telnet to find out:

```
$ telnet some.server.example 25
220 some.server.example ESMTP Exim 3.22 #2 Mon, 14 May 2001 10:24:18 +0100
EHLO client.domain.example
250-some.server.example Hello client.domain.example [192.168.8.20]
250-SIZE 20971520
250-PIPELINING
250-AUTH PLAIN CRAM-MD5
250 HELP
quit
```

If you are setting up an Exim server, you often do not have have much choice about which authentication mechanism to use; in practice, you are stuck with whatever your client software supports. It is, however, worth thinking about the issues.

The main difference between the mechanisms is whether passwords are transmitted in clear or not. How serious an exposure this is depends on the passwords you are using and the networks over which they travel. If the networks are private and secure, or if all the data being transferred is encrypted, this is a less serious concern than if you are using unencrypted connections over the public Internet.

In any case, it is a good idea to use a different set of passwords from the normal login passwords, so that the consequences of disclosure of an SMTP password are limited to potential abuse of mail submission. This is particularly relevant if you require your users to use an encrypted connection for normal logins, but not for SMTP authentication.

Using an alternate password set with CRAM-MD5 authentication means that you do not have to keep normal passwords in clear on the server (just the SMTP passwords); this is probably the safest of the currently supported mechanisms when encryption is not in use. Using encrypted connections, PLAIN or LOGIN are better, because they do not require passwords to be stored in clear on the server.

If you are running a version of Exim that supports SMTP encryption, you can insist that clients using the AUTH command for authentication must start TLS-encrypted sessions first. The availability of the AUTH command is advertised to such hosts only after an encrypted session has been started. To do this, you set auth_over_tls_hosts to match the hosts to which this applies. For example:

```
auth_over_tls_hosts = *
```

means that all authentication must take place over secure sessions. As well as doing this, you need to set up general encryption options. How to do this is described in the section "Encrypted SMTP Connections," later in this chapter:

Exim Authenticators

When Exim is compiled to support SMTP authentication, the seventh part of the runtime configuration file contains settings for a number of *authenticators*. These use a similar syntax to the definitions of routers, directors, and transports. When Exim is receiving SMTP mail, it is acting as a server; when it is sending out messages over SMTP, it is acting as a client. Configuration options are provided for use in both these circumstances, and a single version of Exim can act both as a client and as a server at different times. Each authenticator can have both server and client functions.

To make it clear which options apply to which function, the prefixes server_ and client_ are used on option names that are specific to either the server or the client function, respectively. Server and client functions are disabled if none of their options are set. If an authenticator is to be used for both server and client functions, a single definition, using both sets of options, is required. For example:

```
cram:
  driver = cram_md5
  public_name = CRAM-MD5
  server_secret = ${if eq{$1}{ph10}{secret}fail}
  client_name = ph10
  client_secret = secret2
```

The server_ option is used when Exim is acting as a server, and the client_ options are used when it is acting as a client. An explanation of this example follows later, with the description of the CRAM-MD5 authenticator.

Authentication on an Exim Server

When a message is received from an authenticated host, the value of $received_protocol is set to asmtp instead of esmtp or smtp, and $sender_host_authenticated contains the name of the authenticator driver that successfully authenticated the client. It is empty if there was no successful authentication.

The SMTP AUTH command is accepted from any connected client host. If, however, the client host matches an item in the auth_hosts option, it is *required* to authenticate itself before any commands other than HELO, EHLO, HELP, AUTH, NOOP, RSET, or QUIT are accepted.

A client that matches an item in host_auth_accept_relay is permitted to relay to any domain, provided that it is authenticated, whether or not it matches auth_hosts. In other words, an authenticated client is permitted to relay if it matches either host_accept_relay or host_auth_accept_relay, whereas an unauthenticated client host may relay only if it matches host_accept_relay.

Two common cases are envisaged:

- Certain IP addresses are required to authenticate and are then permitted to relay. This can be handled by setting auth_hosts and either host_accept_relay or host_auth_accept_relay to match the set.
- Any host is permitted to relay, provided it is authenticated. This is handled by setting:

```
    host_auth_accept_relay = *
```

Many variations are possible. For example, in the second case, some hosts could be required to authenticate even for nonrelayed messages by making them match auth_hosts.

Advertising Authentication

If Exim is configured as an authenticating server, it normally advertises this in response to an EHLO command, as described earlier. However, there are circumstances where this is not always wanted. Consider a configuration where some hosts are permitted to relay without authentication by including them in host_accept_relay, whereas others are required to authenticate. What happens when a client host that does not need to authenticate connects? Some client software, on seeing the support for authentication, insists on attempting to authenticate, and there is no way to configure it otherwise. To get round this problem, you can set:

```
auth_always_advertise = false
```

on the server. In this state, authentication support is advertised only if the client host is in auth_hosts or host_auth_accept_relay without being in host_accept_relay.

Testing Server Authentication

Exim's *-bh* option can be useful for testing server authentication configurations (see the section "Testing Incoming Connections," in Chapter 20, *Command-Line*

Interface to Exim). The data for the AUTH command has to be sent encoded in base 64. If you have the *mimencode* command installed on your host, a quick way to produce such data is (for example):

```
echo -n '\0name\0password' | mimencode
```

The command *echo -n* works in most shells (that is, it outputs the following text without a terminating newline). However, in some shells (for example, the Solaris Bourne shell), the *-n* option is not recognized. For these shells:

```
echo '\0name\0password\c' | mimencode
```

often has the desired effect. In the absence of the *mimencode* command, the following Perl script can be used:

```
use MIME::Base64;
printf ("%s", encode_base64(eval "\"$ARGV[0]\""));
```

This interprets its argument as a Perl string and then encodes it. The interpretation as a Perl string allows binary zeros, which are required for some kinds of authentication, to be included in the data. For example, a command line to run this script using the name *encode* might be:

```
encode '\0user\0password'
```

Note, in both examples, the use of single quotes to prevent the shell interpreting the backslashes.

Authenticated Senders

When a client host has authenticated itself, Exim pays attention to the AUTH parameter on incoming SMTP MAIL commands, for example:

```
MAIL FROM:<theboss@acme.com.example> AUTH joker@edu.example
```

The address given in the AUTH parameter is supposed to identify the authenticated original submitter of the message, but this feature does not seem to be in widespread use. The special value <> means "no authenticated sender available." If the client host is not authenticated, Exim accepts the syntax of the AUTH parameter, but ignores the data.

If accepted, the value is available during delivery in the $authenticated_sender variable, and is passed on to other hosts to which Exim authenticates as a client. Do not confuse this value with `$authenticated_id`, which is a string obtained from the authentication process (see later in this chapter), and which is not usually a complete email address.

Authentication by an Exim Client

The **smtp** transport has an option called `authenticate_hosts` when Exim is built with authentication support. When the **smtp** transport connects to a server that announces support for authentication, and also matches an entry in `authenticate_hosts`, Exim (as a client) tries to authenticate as follows:

- For each authenticator that is configured as a client, it searches the authentication mechanisms announced by the server for one whose name matches the public name of the authenticator.
- When it finds one that matches, it runs the authenticator's client code. The variables `$host` and `$host_address` are available for any string expansions that the client might do. They are set to the server's name and IP address. If any expansion is forced to fail, the authentication attempt is abandoned. Otherwise an expansion failure causes delivery to be deferred.
- If the result is a temporary error or a timeout, Exim abandons trying to send the message to the host for the moment. It will try again later. If there are any backup hosts available, they are tried in the usual way.
- If the response to authentication is a permanent error (5*xx* code), Exim carries on searching the list of authenticators. If all authentication attempts give permanent errors, or if there are no attempts because no mechanisms match, it tries to deliver the message unauthenticated.

When Exim has authenticated itself to a remote server, it adds the `AUTH` parameter to the `MAIL` commands it sends if it has an authenticated sender for the message. If a local process calls Exim to send a message, the sender address that is built from the login name and `qualify_domain` is treated as authenticated.

Options Common to All Authenticators

Configured authenticators have names, just like directors, routers, and transports. In addition, there are three options that are common to all authenticators:

`driver` (string, default = unset)
: This option must always be set. It specifies which of the available authenticators is to be used. The currently available values are `plaintext` (which supports both PLAIN and LOGIN authentication) and `cram_md5`.

`public_name` (string, default = unset)
: This option specifies the name of the authentication mechanism that the driver implements, and by which it is known to the outside world. These names should contain only uppercase letters, digits, underscores, and hyphens (RFC 2222), but Exim in fact matches them caselessly. If `public_name` is not set, it defaults to the driver instance's name.

The public names of authenticators that are configured as servers are advertised by Exim when it receives an `EHLO` command, in the order in which they are defined. When an `AUTH` command is received, the list of authenticators is scanned in definition order for one whose public name matches the mechanism given in the `AUTH` command.

`server_set_id` (string, default = unset)

When an Exim server successfully authenticates a client, this string is expanded using data from the authentication, and preserved for any incoming messages in the variable `$authenticated_id`. It is also included in the log lines for incoming messages. For example, a user/password authenticator configuration might preserve the username that was used to authenticate, and refer to it subsequently during delivery of the message.

Using the plaintext Authenticator in a Server

When running as a server, **plaintext** performs the authentication test by collecting one or more data strings from the client, and then expanding a string with the data in $1, $2, and so on. The number of data strings required is controlled by the setting of `server_prompts`, which contains a colon-separated list of prompt strings.

However, the prompts are not necessarily sent as challenges because any strings that are sent with the `AUTH` command are used first. Data supplied on the command line is treated as a list of NUL-separated strings. If there are more strings in `server_prompts` than the number of strings supplied with the `AUTH` command, the remaining prompts are used to obtain more data. Each response from the client may be a list of NUL-separated strings. This general approach allows **plaintext** to be configured to support either PLAIN or LOGIN authentication.

Once a sufficient number of data strings have been received, `server_condition` is expanded. Failure of the expansion (forced or otherwise) causes a temporary error code to be returned. If the result of a successful expansion is an empty string, `0`, `no`, or `false`, authentication fails. If the result of the expansion is `1`, `yes`, or `true`, authentication succeeds and the common `server_set_id` option is expanded and saved in $authenticated_id. For any other result, a temporary error code is returned with the expanded string as the error text.

The PLAIN authentication mechanism (RFC 2595) specifies that three strings be sent with the `AUTH` command. The second and third of them are treated as a user/password pair. Using a single fixed user and password as an example, this could be configured as follows:

```
fixed_plain:
  driver = plaintext
  public_name = PLAIN
```

```
server_condition = ${if and {{eq{$2}{ph10}}{eq{$3}{secret}}}{yes}{no}}
  server_set_id = $2
```

This would be advertised in the response to `EHLO` as:

```
250-AUTH PLAIN
```

and a client host could authenticate itself by sending the command:

```
AUTH PLAIN AHBoMTAAc2VjcmV0
```

The argument string is encoded in base 64, as required by the RFC. This example, when decoded, is `<nul>ph10<nul>secret`, where `<nul>` represents a zero byte. This is split up into three strings, the first of which is empty. The condition checks that the second two are `ph10` and `secret`, respectively. Because no prompt strings are set, if no data is given with the AUTH command, authentication fails.

A more sophisticated instance of this authenticator can make use of the username in $2 to look up a password in a file or database, and maybe do an encrypted comparison (see `crypteq` in chapter Chapter 17, *String Expansion*). For example, if encrypted passwords are available in */etc/passwd*:*

```
server_condition = ${if crypteq{$3}\
  {${extract{1}{:}{${lookup{$2}lsearch{/etc/passwd}{$value}}}}\
  }{yes}{no}}
```

Processes that handle incoming SMTP calls run under the Exim user, so any files that are referenced by the expansion of `server_condition` must be accessible to that user.

For the LOGIN authentication mechanism, no data is sent with the `AUTH` command. Instead, a username and password are supplied separately, in response to prompts. The plaintext authenticator can be configured to support this as in this example:

```
fixed_login:
  driver = plaintext
  public_name = LOGIN
  server_prompts = "User Name : Password"
  server_condition = \
    ${if and {{eq{$1}{ph10}}{eq{$2}{secret}}}{yes}{no}}
  server_set_id = $1
```

This authenticator would in fact accept data as part of the `AUTH` command, but if the client does not supply it (as is the case for LOGIN clients), the prompt strings are used to obtain the two data items.

* Many operating systems no longer keep encrypted passwords directly in */etc/passwd* as implied by this example; also, it is more secure not to use login passwords for SMTP authentication if you can avoid it.

Using plaintext in a Client

The **plaintext** authenticator has just one client option, called `client_send_string`. The string is a colon-separated list of authentication data strings. Each string is independently expanded before being sent to the server. The first string is sent with the `AUTH` command; subsequent strings are sent in response to prompts from the server.

Because the PLAIN authentication mechanism requires zero bytes in the data sent with the `AUTH` command, further processing is applied to each string before it is sent. If there are any single circumflex characters in the string, they are converted to zeros. Should an actual circumflex be required as data, it must be doubled in the string.

This is an example of a client configuration that implements PLAIN authentication mechanism with a fixed name and password:

```
fixed_plain:
  driver = plaintext
  public_name = PLAIN
  client_send = ^ph10^secret
```

The lack of colons in `client_send` means that the entire text is sent with the `AUTH` command, with the circumflex characters converted to zero bytes. A similar example that uses the LOGIN mechanism is:

```
fixed_login:
  driver = plaintext
  public_name = LOGIN
  client_send = : ph10 : secret
```

The initial colon ensures that no data is sent with the `AUTH` command itself. The remaining strings are sent in response to prompts.

Using cram_md5 in a Server

This authenticator has one server option, which must be set to configure the authenticator as a server. It is called `server_secret`. When the server receives the client's response, the "username" is placed in the expansion variable `$1`, and `server_secret` is expanded to obtain the password for that user. The server then computes the CRAM-MD5 digest that the client should have sent, and checks that it received the correct string. If the expansion of `server_secret` is forced to fail, authentication fails. If the expansion fails for some other reason, a temporary error code is returned to the client.

For example, the following authenticator checks that the username given by the client is `ph10`, and if so, uses `secret` as the password. For any other username, authentication fails. A more sophisticated version might look up the secret string in a file, using the username as the key:

```
fixed_cram:
  driver = cram_md5
  public_name = CRAM-MD5
  server_secret = ${if eq{$1}{ph10}{secret}fail}
  server_set_id = $1
```

If authentication succeeds, the setting of `server_set_id` preserves the username in `$authenticated_id`.

Using cram_md5 in a Client

When used as a client, the `cram_md5` authenticator has two options, `client_name` and `client_secret`, which must both be set. They are expanded and used as the username and secret strings, respectively, when computing the response to the server's challenge.

Forced failure of either expansion string is treated as an indication that this authenticator is not prepared to handle this case. Exim moves on to the next configured client authenticator. Any other expansion failure causes Exim to give up trying to send the message to the current server.

A simple example configuration of a `cram_md5` client authenticator, using fixed strings, is as follows:

```
fixed_cram:
  driver = cram_md5
  public_name = CRAM-MD5
  client_name = ph10
  client_secret = secret
```

Most authenticating clients connect only to a single server to deliver their mail, in which case this kind of simple configuration is sufficient. If several servers are involved, the conditional features of expansion strings can be used to select the correct data for each server by referring to $host or $host_address in the options.

Encrypted SMTP Connections

RFC 2487 defines how SMTP connections can be set up so that the data that passes between two hosts is encrypted in transit. Once a connection is established, the client issues a `STARTTLS` command. If the server accepts this, they negotiate an encryption mechanism to be used for all subsequent data transfers. Note that this provides security only when data is in transit between two hosts. It does not provide end-to-end encryption from the original sender to the final mailbox.

Support for Transport Layer Security (TLS), otherwise known as Secure Sockets Layer (SSL), is implemented in Exim by making use of the OpenSSL library.* There is no cryptographic code in the Exim distribution itself.

In order to use this feature you must install OpenSSL, and then build a version of Exim that includes TLS support. You also need to understand the basic concepts of encryption at a managerial level, and in particular, the way that public keys, private keys, and certificates are used, including the concepts of certificate signing and certificate authorities. If you don't understand about certificates and keys, please try to find a source of this background information, which is not specific to Exim or even to mail processing. Some helpful introductory material can be found in the FAQ section for the SSL addition to the Apache web server, at *http://www.modssl.org/docs/2.7/ssl_faq.html#ToC24*

Other parts of this documentation are also helpful, and contain links to further files.

You can create a self-signed certificate using the *req* command provided with OpenSSL, like this:

```
openssl req -x509 -newkey rsa:1024 -keyout file1 -out file2 \
            -days 9999 -nodes
```

file1 and *file2* can be the same file; the key and the certificate are delimited and so can be identified independently. The *-days* option specifies a period for which the certificate is valid; here we are specifying a long time. The `-nodes` option is important: if you do not set it, the key is encryped with a pass phrase that you are prompted for, and any use that is made of the key causes more prompting for the pass phrase. This is not helpful if you are going to use this certificate and key in an MTA, where prompting is not possible.

A self-signed certificate made in this way is sufficient for testing, and may be adequate for all your requirements if you are mainly interested in encrypting transfers and not in secure identification.

Configuring Exim to Use TLS as a Server

When Exim has been built with TLS support, it advertises the availability of the `STARTTLS` command to client hosts that match `tls_advertise_hosts`, but not to any others. The default value of this option is unset, which means that `STARTTLS` is not advertised at all. This default is chosen because it is sensible for systems that want to use TLS only as a client.

* See *http://www.openssl.org/*.

To support TLS on a server, you must set `tls_advertise_hosts` to match some hosts, and you must also specify files that contain a certificate and a private key. For example:

```
tls_advertise_hosts = *
tls_certificate = /etc/secure/exim/cert
tls_privatekey = /etc/secure/exim/privkey
```

The first file contains the server's X509 certificate, and the second contains the private key that goes with it. These files need to be readable by the Exim user. They can be the same file if both the certificate and the key are contained within it.

With just these two options set, Exim will work as a server with clients such as Netscape. It does not require the client to have a certificate (but see the next section for how to insist on this). There is one other option that may be needed in other situations. If `tls_dhparam` is set to a filename, the SSL library is initialized for the use of Diffie-Hellman ciphers with the parameters contained in the file. This increases the set of ciphers that the server supports.*

The strings supplied for the options that specify files are expanded every time a client host connects. It is therefore possible to use different certificates and keys for different hosts, if you so wish, by making use of the client's IP address in $sender_host_address to control the expansion. If a string expansion is forced to fail, Exim behaves as if the option is not set.

Setting Conditions on TLS Connections

If you want to enforce conditions on incoming TLS connections, you must set `tls_verify_hosts` to match the relevant clients. By default, this host list is unset. You could, of course, use the following:

```
tls_verify_hosts = *
```

to make it apply to all TLS connections. When a client host is in this list, two further options are relevant:

- `tls_verify_ciphers` contains a colon-separated list of permitted ciphers. The list is passed to the OpenSSL library, so it must always be colon-separated; Exim's alternate separator feature does not apply. For example:

  ```
  tls_verify_ciphers = DES-CBC3-SHA:IDEA-CBC-MD5
  ```

  With this option set, all TLS sessions must use one of the listed ciphers.

- `tls_verify_certificates` contains the name of a file or a directory that contains a collection of expected certificates. When `tls_verify_certificates` is

* See the command *openssl dhparam* for a way of generating this data.

active, Exim requests a certificate from the client, and fails if one is not provided or does not match any certificate in the collection.

A single file can contain multiple certificates, concatenated end to end. If a directory is used, each certificate must be in a separate file with a name (or a symbolic link) of the form *<hash>.0*, where *<hash>* is a hash value constructed from the certificate. You can compute the relevant hash by running the command:

```
openssl x509 -hash -noout -in /cert/file
```

where */cert/file* contains just one certificate.

Both these options are expanded before use, so again you can make them do different things for different hosts.

Forcing Clients to Use TLS

You can insist that certain client hosts use TLS, by setting `tls_hosts` to match them. When a host is in `tls_hosts`, `STARTTLS` is always advertised to it, even if it is not in `tls_advertise_hosts`. If such a host attempts to send a message without starting a TLS session, the MAIL command is rejected with the error:

```
503 Use of TLS required
```

Allowing Relaying over TLS Sessions

You can permit client hosts to relay, provided they are in a TLS session, by setting `tls_host_accept_relay`. Note that all the host relay checks are alternatives. Relaying is permitted if any of the checks succeed, that is, if any of the following are true:

- The host matches `host_accept_relay`.
- The host is authenticated and matches `host_auth_accept_relay`.
- The host is using a TLS session and matches `tls_host_accept_relay`.

Using `tls_host_accept_relay` probably makes sense only if you are checking the client's certificate, in order to provide some identification.

Variables That Are Set for a TLS Connection

The variable $tls_cipher is set to the name of the cipher that was negotiated for an incoming TLS connection. It is included in the *Received:* header line of an incoming message (by default; you can, of course, change this). It is also included in the log line that records a message's arrival, keyed by `X=`. If you don't want this, set `tls_log_cipher` false.

When Exim has requested a certificate from a client, the value of the Distinguished Name is made available in the variable $tls_peerdn during subsequent processing of the message. Because it is often a long text string, it is not included in the log line or the *Received:* header line by default. You can arrange for it to be logged or keyed by `DN=` by setting `tls_log_peerdn`, and you can use `received_header_text` to change the *Received:* header line.

Configuring Exim to Use TLS as a Client

The `tls_log_cipher` and `tls_log_peerdn` options apply to outgoing SMTP deliveries as well as to incoming messages, the latter option causing logging of the server certificate's Distinguished Name. The remaining client configuration for TLS is all within the **smtp** transport.

It is not necessary to set any options to have TLS work in the **smtp** transport. If TLS is advertised by a server, the **smtp** transport will automatically attempt to start a TLS session. However, this can be prevented by setting `hosts_avoid_tls` (an option of the transport) to a list of server hosts for which TLS should not be used.

If an attempt to start a TLS session fails for a temporary reason (for example, a `4`*`xx`* response to `STARTTLS`), delivery to this host is not attempted. If there are alternative hosts, they are tried; otherwise delivery is deferred. If, on the other hand, the `STARTTLS` command is rejected with a `5`*`xx`* error code, the **smtp** transport attempts to deliver the message in clear, unless the server matches `hosts_require_tls`, in which case delivery is again deferred unless there are other hosts to try.

There are a number of options for the **smtp** transport that match the global TLS options for the server and have the same names:

- `tls_certificate` and `tls_privatekey` provide the client with a certificate, which is passed to the server if it requests it. (If the server is Exim, it will request it only if `tls_verify_certificates` is set.)
- `tls_verify_certificates` and `tls_verify_ciphers` act exactly like their namesakes on the server; they do appropriate verification on the server's certificate and the negotiated cipher, respectively.

These options are all expanded before use with $host and $host_address containing the name and IP address of the server to which the client is connected. Forced failure of an expansion causes Exim to behave as if the relevant option were unset.

SMTP over TCP/IP

SMTP over TCP/IP is the only way of transferring messages between hosts that Exim supports. The next few sections cover some of the detailed processing that occurs. After that, we discuss some other uses of SMTP where a remote host is not involved.

Outgoing SMTP over TCP/IP

Outgoing SMTP over TCP/IP is implemented by the **smtp** transport. If the server's response to `EHLO` indicates that the `SIZE` parameter is supported, Exim adds `SIZE=`*n* to each subsequent `MAIL` command. The value of *n* is the message size plus the value of the `size_addition` option (default 1024), to allow for additions to the message such as per-transport header lines or changes made in a transport filter. If `size_addition` is set negative, the use of `SIZE` is suppressed.

If Exim was built to support SMTP authentication, and the remote server advertises support for the `AUTH` command and also matches `auth_hosts`, Exim scans the authenticator configuration for any suitable client settings. If any are found, it tries to authenticate before sending any messages, as described in the section "SMTP Authentication," earlier in this chapter. However, if no suitable authenticators are found, or if authentication fails, Exim still continues processing and tries to deliver the message regardless.

If a message contains a number of different addresses, all those with the same characteristics (for example, the same envelope sender) that resolve to the same set of hosts in the same order are sent in a single SMTP transaction, even if they are for different domains, unless there are more than the value of the `max_rcpts` option in the **smtp** transport. In this case they are split into groups containing no more than `max_rcpts` addresses each. If `remote_max_parallel` is greater than one, such groups may be sent in parallel sessions. The order of hosts with identical MX values is not significant when checking whether addresses can be batched in this way.

When the **smtp** transport suffers a temporary failure that is not message-related, Exim updates its transport-specific database, which contains records indexed by hostname that remember which messages are waiting for each particular host. It also updates the retry database with new retry times. Exim's retry hints are based on hostname plus IP address, so if one address of a multihomed host is broken, it is skipped most of the time. See Chapter 12, *Delivery Errors and Retrying*, for more detail about error handling and retrying.

When a message is successfully delivered over a TCP/IP SMTP connection, Exim looks in the hints database for the transport to see if there are any queued messages waiting for the host to which it is connected. If it finds one, it creates a new Exim process using the *-MC* option (which can only be used by a process running as root or the Exim user) and passes the TCP/IP socket to it. The new process does only those deliveries that are routed to the connected host, and may in turn pass the socket on to a third process, and so on. When this is happening in a queue run, the queue runner process does not proceed to the next message in the queue until the whole sequence of deliveries is complete.

The `batch_max` option of the **smtp** transport can be used to limit the number of messages sent down a single TCP/IP connection. The second and subsequent messages delivered down an existing connection are identified in the main log by the addition of an asterisk after the closing square bracket of the IP address.

Incoming SMTP Messages over TCP/IP

Incoming SMTP messages over TCP/IP can be accepted in one of two ways: by running a listening daemon or by using *inetd*. In the latter case, the entry in */etc/inetd.conf* should be like this:

```
smtp  stream  tcp  nowait  exim  /usr/exim/bin/exim  in.exim  -bs
```

Exim distinguishes between this case and the case of a local user agent using the *-bs* option by checking whether the standard input is a socket or not.

By default, Exim does not make a log entry when a remote host connects or disconnects (either via the daemon or *inetd*), unless the disconnection is unexpected. It can be made to write such log entries by setting the `log_smtp_connections` option.

The amount of disk space that is available is checked whenever `SIZE` is received on a `MAIL` command, independently of whether `message_size_limit` or `check_spool_space` is configured, unless `smtp_check_spool_space` is set false. A temporary error is given if there is not enough space. The check is for the amount specified in `check_spool_space` plus the value given with `SIZE`, that is, it checks that the addition of the incoming message will not reduce the space below the threshold.

When a message is successfully received, Exim includes the local message ID in its response to the final dot that terminates the data, for example:

```
250 OK id=13M6GM-0005kt-00
```

If the remote host logs this text, it can help with tracing what has happened to a message if a query is raised.

Exim can be configured to verify addresses in incoming SMTP commands as they are received. See Chapter 13 for details. It can also be configured to rewrite addresses at this time, before any syntax checking is done. See Chapter 14, *Rewriting Addresses*.

The VRFY and EXPN Commands

RFC 821 defines two SMTP commands that were intended to be helpful aids in debugging delivery problems: `VRFY` verifies an email address, and `EXPN` lists the expansion of an alias or mailing list. In former times, when the Internet was a friendlier place where messages were often delivered directly to their destination hosts, mail administrators made use of these commands regularly. Nowadays, with so much more mail being delivered indirectly via mail hubs and gateways, their potential usefulness has declined, and in addition, many administrators regard them as security exposures.

Exim supports neither of these commands by default. `VRFY` is permitted only when the configuration option `smtp_verify` is explicitly set. Otherwise, it responds in this way:

```
252 VRFY not available
```

A success code (252) rather than an error code is used because some broken clients issue a `VRFY` command before attempting to send a message. When `VRFY` is accepted, exactly the same code as when Exim is called with the *-bv* option is run.

`EXPN` is permitted only if the calling host matches `smtp_expn_hosts` (add `localhost` if you want calls to the loopback address to be able to use it). A single-level expansion of the address is done. `EXPN` is treated as an address test (similar to the *-bt* option) rather than a verification (the *-bv* option). If an unqualified local part is given as the argument to `EXPN`, it is qualified with `qualify_domain`.

Rejections of `VRFY` and `EXPN` commands are logged on the main and reject logs, and `VRFY` verification failures are logged on the main log for consistency with `RCPT` failures.

The ETRN Command

RFC 821 describes a command called `TURN`, which reverses the roles of the client and server. The idea was that a client could connect, send its outgoing mail, and then use `TURN` to become the server to receive incoming messages. However, because no authentication was defined in RFC 821, this has serious security problems.

RFC 1985 describes an SMTP command called `ETRN` that is intended to overcome the security problems of the original `TURN` command, while still permitting a client to connect and request that pending mail be delivered. This has found some favor in communities where clients connect to servers by dial-up methods.

The ETRN command is concerned with "releasing" messages that are awaiting delivery to certain hosts. They are not sent down the same connection that issued the ETRN command, but are routed in the normal way over fresh TCP/IP connections, thus avoiding the security problems of TURN. Exim contains support for ETRN, but it does not fit naturally into the way Exim is designed. Because Exim does not organize its message queue by host, it is not straightforward to find "all messages waiting for this host." If you run a server that is a holding system for dial-up systems, and there is more than a trivial amount of mail to be kept, you should consider delivering the pending mail into local files, using a different directory for each host, say, as discussed in the section "Intermittently Connected Hosts," in Chapter 12. ETRN can still be used to start up a delivery program that reads messages from these files.

The ETRN command can be used in several formats in which its argument is defined to be a host or a domain name. The only form that is supported entirely within Exim is the one where the text starts with the # prefix, in which case the interpretation of the remainder of the text is not defined and is specific to the SMTP server. A valid ETRN command causes a run of Exim with the *-R* option, with the remainder of the ETRN text as its argument. For example:

```
ETRN #brigadoon
```

runs the command:

```
exim -R brigadoon
```

which causes a delivery attempt on all messages with undelivered addresses containing the text `brigadoon`. All addresses in the messages are considered for delivery, not just the ones that trigger the selection. Note that the supplied string is not necessarily a host or domain name.

Exim recognizes ETRN only if the calling host matches `smtp_etrn_hosts`, an option that is unset by default. Attempts to use ETRN from other hosts are logged on the main and reject logs; when ETRN is accepted, it is logged on the main log.

When `smtp_etrn_serialize` is set (the default), it prevents the simultaneous execution of more than one queue run for the same argument string as a result of an ETRN command. This stops a misbehaving client from starting more than one queue runner at once. Exim implements the serialization by means of a hints database in which a record is written whenever a process is started by ETRN, and deleted when a *-R* queue run completes.

Obviously, there is scope for hints records to be left lying around if there is a system or program crash. To guard against this, Exim ignores any records that are more than six hours old, but you should normally arrange to delete any files in the *spool/db* directory whose names begin with `serialize-` after a system reboot.

For more control over what `ETRN` does, the `smtp_etrn_command` option can be used. This specifies a command that is run whenever `ETRN` is received, whatever the form of its argument. For example:

```
    smtp_etrn_command = /etc/etrn_command $domain $sender_host_address
```

The string is split up into arguments that are independently expanded. The variable $domain is set to the argument of the ETRN command, but no syntax checking is done on the contents of this argument. A new freestanding process is created to run the command. Exim does not wait for it to complete, so its status code is not checked. As Exim is normally running under its own uid and gid when receiving incoming SMTP, it is not possible for it to change them before running the command.

Important: If you use `smtp_etrn_command` to do something other than run Exim with the *-R* option, you must disable `smtp_etrn_serialize`, because otherwise hints are never deleted, and further `ETRN` commands are ignored until the hints time out.

Local SMTP

Some user agents use SMTP to pass messages to their local MTA using the standard input and output, as opposed to passing the envelope on the command line and writing the message to the standard input. This is supported by the *-bs* command-line option. This form of SMTP is handled in the same way as incoming messages over TCP/IP, except that all host-specific processing is bypassed, and any envelope sender given in a `MAIL` command is ignored unless the caller is trusted.

Conversely, some software applications for managing message stores accept incoming messages from an MTA using a variation of SMTP known as LMTP (RFC 2033). Exim supports this either via the **lmtp** transport for communicating with a local process over a pipe, or by the `protocol` option of the **smtp** transport for using LMTP over TCP/IP (see the section "The lmtp Transport" and the section "The smtp Transport," respectively, in Chapter 9, *The Transports*).

Batched SMTP

Batched SMTP is a format for storing mail messages, in which the envelopes are prepended to the message in the form of SMTP commands. It is mostly used as an intermediate format between Exim and another form of transport such as UUCP, or as a private delivery agent for dial-up clients. Delivering messages into files in batched SMTP format is discussed in Chapter 9.

Messages from other sources that are in batch SMTP format can be passed to Exim by means of the *-bS* command-line option, which causes Exim to accept one or more messages by reading SMTP on the standard input, but generate no responses. If the caller is trusted, the senders in the `MAIL` commands are believed; otherwise, the sender is always the caller of Exim. Unqualified senders and receivers are not rejected (there seems little point) but instead are automatically qualified. If `sender_verify` is set, sender verification takes place only if `sender_verify_batch` is set (it defaults unset). Receiver verification and administrative rejection is not done, even if configured. `HELO` and `EHLO` act as `RSET`; `VRFY`, `EXPN`, `ETRN`, `HELP`, and `DEBUG` act as `NOOP`; `QUIT` quits.

If any error is detected while reading a message, including a missing dot at the end, Exim gives up immediately. It writes details of the error to the standard output in a stylized way that the calling program should be able to make some use of automatically, for example:

```
554 Unexpected end of file
Transaction started in line 10
Error detected in line 14
```

It writes a more verbose version for human consumption to the standard error file, for example:

```
An error was detected while processing a file of BSMTP input.
The error message was:

  501 '>' missing at end of address

The SMTP transaction started in line 10.
The error was detected in line 12.
The SMTP command at fault was:

   rcpt to:<malformed@in.com.plete

1 previous message was successfully processed.
The rest of the batch was abandoned.
```

The return code from Exim is zero only if there were no errors. It is 1 if some messages were accepted before an error was detected, and 2 if no messages were accepted.

16

File and Database Lookups

We have introduced and given brief explanations of the way Exim can be configured to look up data in files and databases in earlier chapters. Lookups give you a lot of flexibility in the way you store the data that controls Exim's behavior. They also allow Exim to make use of common, companywide databases that can be shared with other programs. Lookups can be used in several different kinds of configuration items, but they operate in the same way in each case. This chapter covers the underlying lookup mechanisms in detail; there are many examples of lookup usage throughout the book.

You can specify lookups in three different types of configuration items:

- Any string that is to be expanded may contain explicit lookup requests, which make it possible to replace portions of the string by data read from a file or database. Details of the use of lookups in string expansions are given in Chapter 17, *String Expansion*.
- The **aliasfile** director and the **domainlist** router can be configured directly to look up data in files.
- A number of configuration options contain lists of domains, hosts, or mail addresses that are to be checked against some item of data related to a message. These lists can contain lookup items as a way of avoiding excessively long linear lists in the configuration file. The item being checked is used as a key for the lookup, and if the lookup succeeds, the item is taken as being in the list. For long lists, an indexed lookup gives much improved performance over a linear scan. When Exim is using a lookup to check whether something

is in a list, any data that is returned by the lookup is discarded; whether the lookup succeeds or fails is all that counts.* Details of the use of lookups in lists are given in Chapter 18, *Domain, Host, and Address Lists*.

There are a number of different types of lookups, and each is implemented by a separate module of code that is included in Exim only if it is requested when the binary is built. This makes it easy to add new kinds of lookups, while at the same time not requiring every Exim binary to include all possible lookups. The default build-time configuration includes only the `lsearch` and `dbm` lookups, so if you want to use any other kind, you must ensure that Exim has been built to include them.

There are two different ways in which the main part of Exim can call a lookup module, but each individual module uses just one of them. This is known as the *style* of the lookup.

- Modules that support the *single-key* style are given the name of a file in which to look, and a key to search for. The lookup type determines how the file is searched. If the key is found in the file, a single data string is returned. This style handles plain files, indexed files, and NIS maps.
- Modules that support the *query* style accept a generalized query, which may contain one or more items of data, and they may return one or more items of data. This style is used to access databases such as those held by NIS+, LDAP, MySQL, or PostgreSQL.

To perform a lookup, Exim hands over the necessary data to the relevant lookup module, and receives in return a string of data that was looked up, or an indication that the lookup did not succeed. Each lookup, even when coded entirely within Exim itself, is treated as a "black box" whose internals are not visible to the rest of the program. This is illustrated in Figure 16-1.

There are no restrictions on where the different kinds of lookup can be used. Any kind of lookup can be used wherever a lookup is permitted.

Single-Key Lookup Types

A single-key lookup provides a way of searching a set of data consisting of (*key*, *value*) pairs, where the keys are fixed strings. Such data is often shown as a sequence of lines with a key at the start of each line, separated from its data by a

* This is a simplification; there are in fact two special cases, the `domains` and `local_parts` options in directors and routers, where the data is preserved for later use.

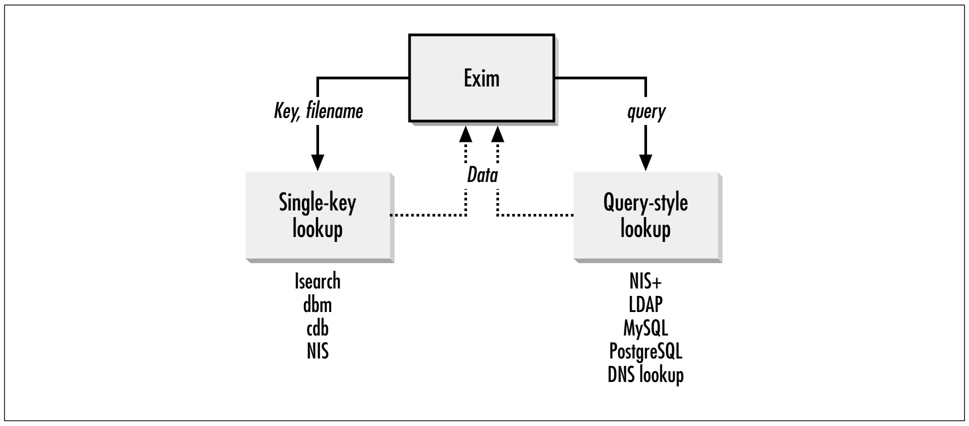


Figure 16-1. File and database lookups

colon, even if in operation the data is actually stored in some other way, such as in a NIS map. For example:

```
root:          postmaster@simple.example,
postmaster:    simon@simple.example
```

A file that contains data in this format can in fact be searched directly by means of the `lsearch` lookup type, which generalizes the format slightly. The file is searched linearly from the start for a line beginning with the key, and terminated by a colon or whitespace or the end of the line. Whitespace between the key and the colon is permitted (and ignored). The remainder of the line, with leading and trailing whitespace removed, is the data. This can be continued onto subsequent lines by starting them with any amount of whitespace, but only a single-space character is retained in the data at such a junction. If the data begins with a colon, the key must be terminated by a colon rather than whitespace, for example:

```
exuser:  :fail: This person has gone away.
```

Empty lines and lines beginning with # are ignored, even if they occur in the middle of an item. This is the traditional format of alias files, and on most systems, an example can be seen in */etc/aliases*. The keys in `lsearch`ed files are literal strings and are not interpreted in any way.

When such files are large, searching them linearly is inefficient, and it is better to convert the data into one of the other single-key formats, in which an index is used for faster lookup. It is not necessary to know the detailed format of such files, as they are read and written only by the library functions that implement the lookup method.

The most common such format is called DBM. Most modern versions of Unix have a DBM library installed as standard, though this is not true of some older systems. The two most common DBM libraries are *ndbm* (standard on Solaris and IRIX) and Berkeley DB Version 2 or 3 (standard on several free operating systems).* Exim supports both of these, as well as the older Berkeley DB Version 1, *gdbm*, and *tdb*. The choice of which DBM library to use is made when Exim is built.

Because DBM is so commonly used, Exim comes with a utility called *exim_dbmbuild*, which creates a DBM file† from a file in traditional alias format. For example:

```
exim_dbmbuild /etc/aliases /etc/aliases.db
```

builds a DBM version of the system aliases file and calls it */etc/aliases.db*. This can then be used by changing the search type and filename in the director that handles aliases, so that instead of a linear search, a keyed DBM lookup is now used:

```
system_aliases:
  driver = aliasfile
  file = /etc/aliases.db
  search_type = dbm
```

One complication in the implementation of DBM lookups is whether the key strings include a terminating binary zero byte or not. Both Exim itself and the *exim_dbmbuild* utility include a terminating zero. However, if you use some other means of creating DBM files (for example, the DBM functions in Perl), you may end up with files that do not have this extra character in the keys. Such files can be read by Exim using the `dbmnz` lookup type instead of `dbm`.

Using a DBM file is more efficient than a linear search if the file has more than a few dozen entries in it. However, because DBM libraries provide both reading and updating facilities, it is not as efficient as a read-only indexed file format, which requires less overhead. This is where the next lookup type comes in.

The `cdb` lookup type searches a Constant DataBase file. The cdb format is designed for indexed files that are read frequently and never updated, except by total re-creation. As such, it is particulary suitable for large files containing aliases or other indexed data referenced by an MTA. Information about cdb is found at *http://www.pobox.com/~djb/cdb.html*. The cdb distribution isn't needed to build Exim with cdb support, because the code for reading cdb files is included directly in Exim itself. However, no means of building or testing cdb files is provided with Exim because these are available within the cdb distribution. The usual way to build a cdb file from a "flat" file is to run a command of the following form:

```
12tocdbm < /etc/aliases | \
  cdbmake /etc/aliases.cdb /etc/aliases.tmp
```

* You can find information about Berkeley DB at *http://www.sleepycat.com*.

† Or files, because some DBM libraries use more than one file.

This uses two utilities that are part of the cdb distribution. However, it works only on input files that have spaces separating the keys and data (a colon is not treated specially), and no continuation lines, so conventional alias files may need editing first. The *12tocdbm* command converts the flat file into a format that has the lengths of the key and data at the start of each line, and the *cdbmake* command uses this data to create a cdb file in the temporary file whose name is its second argument. If this is successful, the temporary file is renamed to the first argument.

The final type of single-key lookup is `nis`. This does a NIS lookup using the filename as the name of a NIS map in which to look up the key.* Exim doesn't recognize aliases for NIS map names; the full name must always be used, as in this director:

```
system_aliases:
  driver = aliasfile
  file = mail.aliases
  search_type = nis
```

Query-Style Lookup Types

Query-style lookups give access to collections of data where the search function processes a generalized query, as opposed to the simple filename and key string that the single-key lookups use. In addition, some query-style lookups can return more than one value at once. You can pick out individual values from such data using the `extract` feature of string expansions.†

The query-style lookup interface allows Exim to read data from a number of general database servers, brief descriptions of which are included here. Before using any of these databases with Exim, you should first become familiar with the concepts and mode of operation from the database's own documentation.

Quoting Lookup Data

When data from an incoming message is included in a query-style lookup, special characters in the data can cause syntax errors in the query. For example, a NIS+ query that contains:

```
[name=$local_part]
```

will be broken if the local part happens to contain a closing square bracket.

* The terminating zero is normally excluded from the key, but there is a variant called `nis0` that does include the terminating binary zero in the key.

† See the section "Extracting Fields from Substrings," in Chapter 17.

For NIS+, data can be enclosed in double quotes in the following manner:

```
[name="$local_part"]
```

but this then leaves the problem of a double quote in the data. The rule for NIS+ is that double quotes must be doubled. Other lookup types have different rules, and to cope with the differing requirements, an expansion operator is provided for each query-style lookup to quote according to the lookup's rules. For example, the safe way to write the NIS+ query is:

```
[name="${quote_nisplus:$local_part}"]
```

See Chapter 17 for full coverage of string expansions. A quote operator can be used for all lookup types, but has no effect for single-key lookups, because no quoting is ever needed in their key strings.

NIS+

Although NIS+ does not seem to have become as popular as NIS, it is nevertheless used by some sites. Using NIS+ jargon, a query consists of an *indexed name* followed by an optional colon and field name. If a field name is included, the result of a successful query is the contents of the named field; if a field name is not included, the result consists of a concatenation of *field-name=field-value* pairs, separated by spaces. Empty values and values containing spaces are quoted. For example, the following query:

```
[name=mg1456],passwd.org_dir
```

might return the string:

```
name=mg1456 passwd="" uid=999 gid=999 gcos="Martin Guerre"
home=/home/mg1456 shell=/bin/bash shadow=""
```

(split over two lines here to fit on the page), whereas:

```
[name=mg1456],passwd.org_dir:gcos
```

would just return the gcos field as

```
Martin Guerre
```

with no quotes. A NIS+ lookup fails if NIS+ returns more than one table entry for the given indexed name. An example of a director that uses NIS+ to handle aliases is:

```
system_aliases:
  driver = aliasfile
  query = [name="${quote_nisplus:$local_part}"],aliases.org_dir:address
  search_type = nisplus
```

Instead of the `file` option that is used for single-key lookups, the `query` option is used. The data that is being looked up (the local part) is given explicitly in the query, whereas for single-key lookups it is implicit. This query assumes there is a NIS+ table called *aliases.org_dir* containing at least two fields, *name* and *address*.

LDAP

An increasingly popular protocol for accessing databases that hold information about users is LDAP. Exim supports LDAP queries in the form of a URL, as defined in RFC 2255. For example, in the configuration of an `aliasfile` director, one might have these settings:

```
system_aliases:
  driver = aliasfile
  search_type = ldap
  query = ldap:///\
          cn=${quote_ldap:$local_part},o=University%20of%20Cambridge,\
          c=UK?mailbox?base?
```

This searches for a record whose `cn` field is the local part, and extracts its mailbox field. This example does not specify which LDAP server to query, but a specific LDAP server can be specified by starting the query with:

```
ldap://hostname:port/...
```

If the port (and preceding colon) are omitted, the standard LDAP port (389) is used.

When no server is specified in a query, a list of default servers is taken from the `ldap_default_servers` configuration option. This supplies a colon-separated list of servers that are tried in turn until one successfully handles a query or there is a serious error. Successful handling either returns the requested data or indicates that it does not exist. Serious errors are syntactical, or finding multiple values when only a single value is expected. Errors that cause the next server to be tried are connection failures, bind failures, and timeouts. In other words, the servers in the list are expected to contain identical data, with the later ones acting as back-ups for the earlier ones.

For each server name in the list, a port number can be given. The standard way of specifying a host and port is to use a colon separator (RFC 1738). Because `ldap_default_servers` is a colon-separated list, such colons have to be doubled. For example:

```
ldap_default_servers = \
  ldap1.example.com::145 : ldap2.example.com
```

If `ldap_default_servers` is unset, a URL with no server name is passed to the LDAP library with no server name, and the library's default (normally the local host) is used.

The LDAP URL syntax provides no way of passing authentication and other control information to the server. To make this possible, the URL in an LDAP query may be preceded by any number of *name=value* settings, separated by spaces. If a value contains spaces, it must be enclosed in double quotes. The following names are recognized:

user
: Sets the Distinguished Name for authenticating the LDAP bind.

pass
: Sets the password for authenticating the LDAP bind.

size
: Sets the limit for the number of entries returned.

time
: Sets the maximum waiting time for a query.

The values may be given in any order. Here is the `query` option of the previous example with added authentication data:

```
query = \
  user="cn=admin,o=University of Cambridge,c=UK" \
  pass = secret \
  ldap:///\
  cn=${quote_ldap:$local_part},o=University%20of%20Cambridge,\
  c=UK?mailbox?base?
```

A problem with placing a password directly in a query such as this is that the values of Exim's configuration settings can be obtained by using the *-bP* command-line option, so any user of the system who can run Exim can see this information. If you are using Exim 3.20 or later, you can prevent this by putting the word `hide` in front of the option setting:

```
hide query = ...
```

When `hide` is present, only admin users can extract the value using *-bP*. Another way of keeping the password secret is to place it in a separate file, accessible only to the Exim user, and use a lookup in the expansion to obtain its value.

Data Returned by an LDAP Lookup

Although in many cases, such as the earlier example, an LDAP query is expected to find a single entry and extract the value of just one of its attributes, an LDAP lookup may in fact find multiple entries in the database, each with any number of attributes. However, if a lookup finds an entry with no attributes, it behaves as if the entry did not exist.

You can control what happens if more than one entry is found by varying the lookup type. There are three LDAP lookup types that behave differently in the way they handle the results of a query:

`ldap`
: `ldap` requires the result to contain just one entry; if there are more, it gives an error. However, more than one attribute value may be taken from the entry.

`ldapdn`
: `ldapdn` also requires the result to contain just one entry, but it is the Distinguished Name that is returned rather than any attribute values.

`ldapm`
: `ldapm` permits the result to contain more than one entry; the attributes from all of them are returned, with a newline inserted between the data from each entry.

In the common case where you specify a single attribute in your LDAP query, the result is not quoted, and if there are multiple values, they are separated by commas. If you specify multiple attributes, they are returned as space-separated strings, quoted if necessary, and preceded by the attribute name. For example:

```
ldap:///o=base?attr1,attr2?sub?(uid=fred)
```

might yield:

```
attr1="value one" attr2=value2
```

If you do not specify any attributes in the search, the same format is used for all attributes in the entry. For example:

```
ldap:///o=base??sub?(uid=fred)
```

might yield:

```
objectClass=top attr1="value one" attr2=value2
```

The `extract` operator in string expansions can be used to pick out individual fields from such data.

MySQL and PostgreSQL

MySQL and PostgreSQL are open source database packages whose queries are expressed in the SQL language. Handling aliases using MySQL could be configured like this:

```
system_aliases:
  driver = aliasfile
  query = select mailbox from userdata \
          where id='${quote_mysql:$local_part}'
  search_type = mysql
```

For PostgreSQL, the configuration is identical, except that `mysql` is replaced by `pgsql` wherever it appears.

If the result of the query contains more than one field, the data for each field in the row is returned, preceded by its name, so the result of the following:

```
select home,name from userdata where id='ph10'
```

might be:

```
home=/home/ph10 name="Philip Hazel"
```

Values containing spaces and empty values are double-quoted, with embedded quotes escaped by backslash.

If the result of the query contains just one field, the value is passed back verbatim without a field name, for example:

```
Philip Hazel
```

If the result of the query yields more than one row, it is all concatenated, with a newline between the data for each row.

Before it can use MySQL or PostgreSQL lookups, Exim has to be told where the relevant servers are. There is no default server as for LDAP. It also needs to know which username and password to use when connecting to a server, and which database to search. This is done by setting the `mysql_servers` or `pgsql_servers` option to a colon-separated list of data for each server. Each item contains a host-name, database name, username, and password, separated by slashes. This example lists two servers:

```
hide pgsql_servers = localhost/userdb/root/secret:\
                     otherhost/userdb/root/othersecret
```

Note the use of the `hide` prefix to protect the value of the variable. When `hide` is present, only admin users can extract the value using *-bP*. For each query, the servers are tried in order until a connection and a query succeeds. A host may be specified as *`name`*:*`port`*, but because this is a colon-separated list, the colon has to be doubled.

For MySQL, the database name may be empty, for example:

```
hide mysql_servers = localhost//root/secret
```

but if it is, the identity of the database must be specified in every query. For example:

```
select mailbox from userdb.userdata where ...
```

is a query that selects from the table called `userdata` in the database called `userdb`. This facility is not available in PostgreSQL.

DNS Lookups

Direct access to the DNS is available within Exim's configuration through a query-style lookup called `dnsdb`. A query consists of a DNS record type name and a DNS domain name, separated by an equal sign. The result of a lookup is the righthand side of any DNS records that are found, separated by newlines if there are more than one of them, in the order they were returned by the DNS resolver. For example, this string expands into the MX hosts for the domain *a.b.example*:

```
${lookup dnsdb{mx=a.b.example}{$value}}
```

The DNS types that are supported are A (IPv4 address), AAAA and A6 (IPv6 address, available when Exim is compiled with IPv6 support), CNAME (canonical name), MX (mail exchanger), NS (name server), PTR (pointer), and TXT (text). When the type is MX, the data for each record consists of the preference value and the hostname, separated by a space. When the type is PTR, the address should be given as normally written; it is converted to the necessary *in-addr.arpa* (IPv4) or *ip6.arpa* (IPv6) format internally. For example:

```
${lookup dnsdb{ptr=192.168.4.5}{$value}}
```

If the type is omitted, it defaults to TXT for backwards compatibility with earlier versions of Exim.

Implicit Keys in Query-Style Lookups

When a query-style lookup is used in a string expansion or a driver configuration (such as the earlier examples), all the parameters of the lookup are specified explicitly, and the data that is needed to construct them is available in Exim's expansion variables.

However, when a lookup appears in a host, domain, or address list, there is an implicit key that does not necessarily correspond to the value of any variable. This is not a problem for single-key lookups; the relevant filename is specified, and the implicit key is used. For example, the list of local domains could be given as:

```
local_domains = dbm;/local/domain/list
```

which checks for a local domain by doing a DBM lookup on the file */local/domain/list*. If there is a record in the file whose key matches the domain that is being checked, the domain is treated as local.

However, with query-style lookups, a complete query has to be specified. To do this, some means of including the implicit key is required because each query-style lookup has its own query syntax. The special expansion variable $key is provided for this purpose when lists are being scanned. NIS+ could be used to look up local domains by a setting such as:

```
    local_domains = nisplus;[domain=$key],domains.org_dir
```

Because of the need to incorporate the value of `$key`, query-style lookups that appear in lists are expanded before they are used.

Temporary Errors in Lookups

Lookup functions can return temporary error codes if the lookup cannot be completed. For example, an LDAP or NIS database might be unavailable. When this occurs in a transport, director, or router, delivery of the message is deferred, as for any other temporary error. In other circumstances, Exim generates a temporary error if possible. It is not advisable to use a lookup that might defer for critical options such as `local_domains`.

Default Values in Single-Key Lookups

In this context, a "default value" is a value specified by the administrator to be used instead of the result of a lookup if the lookup fails to find any data for the given key. If an asterisk is added to a single-key lookup type name (for example, `lsearch*`), and the initial lookup fails, the key that consists of the literal string `*` is looked up in the file to provide a default value. For example, if an alias file contains:

```
    *:           info
    postmaster:  pat
    sales:       sam
```

any local part other than *postmaster* or *sales* is aliased to *info*. Notice that putting the default entry first does not mean that it applies to every lookup. The asterisk used here is not a wildcard in the sense that it matches anything; it is just a convenient string to use to mean "default." In linearly searched files, putting it first means that it is found quickly when it is needed.

However, this example would not actually be a useful alias file in a configuration where aliasing is handled by the first director (as in the default configuration). Addresses generated by aliasing are reevaluated, so when *pat* and *sam* are directed, the alias default is taken, and so all messages would end up in the *info* mailbox. The simplest way to solve this problem is to put the aliasing director last instead of first.

When alias files have keys that are complete mail addresses rather than just local parts, defaults for individual domains may be needed. Consider this file:

```
    ted@domain1.example:  edward@domain3.example
    ted@domain2.example:  edwin@domain4.example
    *@domain1.example:    postmaster@domain2.example
    *:                    postmaster@domain3.example
```

It provides a default for *domain1.example* and a general default for all other domains. Such a file can be handled by adding `*@` to a single-key lookup type name (for example `dbm*@`). If the initial lookup fails and the key contains an @ character, a second lookup is done with everything before the last `@` in the key replaced by `*`. If the second lookup fails (or doesn't take place because there is no `@` in the key), `*` is looked up as an ultimate default. If the original key is *jim@domain1.example*, for instance, a single-key search type such as `cdb*@` would look up the following keys:

```
jim@domain1.example
*@domain1.example
*
```

The first one that succeeds provides the result of the lookup.

Partial Matching in Single-Key Lookups

Normally, a single-key lookup searches the file for an exact match with the given key. However, in a number of situations where domains are being looked up, it is useful to be able to do partial matching, where only the final components of the search key match a key in the file. Suppose, for instance, you want to match all domains that end in *dates.example*. First, you must create the special key:

```
*.dates.example
```

in the file. No normal search for a domain can ever match that item, because domains cannot contain asterisks. However, if you add the string `partial-` to the start of a single-key lookup type (for example, `partial-dbm`), Exim behaves in a different way. First, it looks up the original key; if that fails, an asterisk component is added at the start of the subject key, and it is looked up again. If that fails, Exim starts removing dot-separated components from the start of the subject key, one-by-one, and adding an asterisk component on the front of what remains. This means that keys in the file that do not begin with an asterisk are matched only by unmodified subject keys even when partial matching is in use.

A default minimum number of two nonasterisk components is required. If the subject key is *2250.future.dates.example*, Exim does the following lookups, in this order:

```
2250.future.dates.example
*.2250.future.dates.example
*.future.dates.example
*.dates.example
```

As soon as one key in the sequence succeeds, the lookup finishes. The minimum number of components can be adjusted by including a number before the hyphen in the search type. For example, `partial3-lsearch` specifies a minimum of three nonasterisk components in the modified keys. If `partial0-` is used, the original key is shortened right down to the null string, and the final lookup is for an asterisk on its own.

If the search type ends in `*` or `*@` (see the section "Default Values in Single-Key Lookups," earlier in this chapter), the search for an ultimate default that this implies happens after all partial lookups have failed. If `partial0-` is specified, adding `*` to the search type has no effect, because the `*` key is already included in the sequence of partial lookups.

The use of * in lookup partial matching differs from its use as a wildcard in domain lists and the like, where it just means "any sequence of characters." Partial matching works only in terms of dot-separated components, and the asterisk in a partial matching subject key must always be followed by a dot. For example, you could not use the following:

```
*key.example
```

as a key in a file to match partial lookups of *donkey.example* and *monkey.example*.

Lookup Caching

An Exim process caches the most recent lookup result on a per-file basis for single-key lookup types, and keeps the relevant files open. In some types of configuration, this can lead to many files being kept open for messages with many recipients. To avoid hitting the operating system limit on the number of simultaneously open files, Exim closes the least recently used file when it needs to open more files than its own internal limit, which can be changed via the `lookup_open_max` option. For query-style lookups, a single data cache per lookup type is kept.

17 *String Expansion*

The combination of expansions and lookups makes it possible to configure Exim in many different ways. If you want to explore these different possibilities, you need to understand what string expansions can do for you. We cover a number of examples in earlier chapters; this chapter contains a full explanation of the mechanism, and descriptions of all the different expansion items. A reference summary of string expansion, including a list of all the expansion variables, is given in Appendix A, *Summary of String Expansion*.

When Exim is expanding a string, special processing is triggered by the appearance of a dollar sign. The expander copies the string from left to right until it hits a dollar, at which point it reads to the end of the expansion item, does whatever processing is required, and adds the resulting substring to its output before continuing to read the rest of the original string. Most, but not all, expansion items involve the use of curly brackets (braces) as delimiters. For example, when expanding the following string:

```
    Before-${substr_4_2:$local_part}-After
```

the expander copies the initial substring `Before-`, then processes the expansion item `${substr_4_2:$local_part}` to produce the next part of the result, and finally adds the substring `-After` at the end.

To make it possible to include a dollar character in the output of an expansion, backslash is treated as an escape character by the expander. A number of special sequences, such as `\n` for newline, are recognized.* If any other character follows a backslash, it is treated as a literal with no special meaning. Thus, a literal dollar sign is entered as `\$` and a literal backslash as `\\`. Dollars and backslashes are

* These are in fact exactly the same set as those that are recognized by the string-reading code.

almost always associated with regular expressions when they appear in expanded strings. For example, the regular expression:

```
^\d{8}@example\.com$
```

matches an email address where the local part consists of precisely eight digits, and the domain is *example.com*. Suppose that you wanted to place the mailboxes for such users in a special directory. Whereas an ordinary user's mailbox is:

```
/var/mail/$local_part
```

you want to arrange for this group of users' mailboxes to be:

```
/var/mail/special/$local_part
```

You could use a value like this for the `file` option in an **appendfile** transport:

```
file = /var/mail/\
  ${if match {$local_part}{^\d{8}@example\.com$}\
  {special/}}$local_part
```

The details of how the `if` expansion item works are described later in the section on conditionals, but the idea is to match the local part against the regular expression, and if it matches, to insert `special/` into the filename.

However, the previous option setting does not work because when the string is expanded, the dollar and backslashes that are part of the regular expression are interpreted by the expander before it attempts the regular expression match, and an error occurs, because `$}` is not a valid expansion item. So we have to use:

```
file = /var/mail/\
  ${if match {$local_part}{^\\d{8}@example\\.com\$}\
  {special/}}$local_part
```

where an extra backslash has been inserted before every dollar and backslash in the regular expression.*

In the following sections, we cover the various different kinds of items that can occur in expansion strings. Whitespace may be used between subitems that are keywords, or between substrings enclosed in braces inside an outer set of braces to improve readability, but any other whitespace is taken as part of the string.

* If you enclose the value in double quotes, you have to insert yet more backslashes because they are also special inside double quotes. This is a good reason for avoiding quotes unless you really need their backslash interpretation.

Variable Substitution

We use the variable $local_part many times in configuration examples, and also mention several other variables. In fact, a large number of variables exist, containing data that might be of use in configuration files, and also in system and user filters. A complete list is given in Appendix A.

In most cases, we have been able to reference $local_part by including:

```
$local_part
```

in the configuration settings we have used. This format is fine, provided that what follows the variable name is not a letter, digit, or underscore. If it is, the alternative syntax:

```
${local_part}
```

must be used, so that the end of the variable's name can be distinguished. An expansion syntax error occurs if the name is unknown to Exim. You can test whether a variable contains any data by means of the `def` condition, which is described later in this chapter.

Header Insertion

The contents of a specific message header line can be inserted into a string by an item of the form:

```
$header_from:
```

The abbreviation `$h` can be used instead of `$header`, and header names are not case-sensitive. This example inserts the contents of the *From:* header line. It is similar to the insertion of the value of a variable, but there are some important differences:

- The name must be terminated by a colon, and curly brackets must *not* be used because they are permitted in header names.* The colon is not included in the expanded text. If the name is followed by whitespace, the colon may be omitted; in this case the whitespace *is* included in the expanded text. However, it is best to get into the habit of including the colon, so that you don't leave it out when it is really needed.
- If the message does not contain a header with the given name, the expansion item adds nothing to the string; it doesn't fail. Thus, use of the correct spelling of header names is vital. If you use `$header_reply_to:`, for example, it doesn't

* Any printing characters except colon and whitespace are permitted by RFC 822.

insert the contents of the *Reply-To:* header. You can test for the presence of a particular header by means of the `def` condition, which is described later in this chapter.

The contents of header lines are most frequently referenced from filter files (see Chapter 10, *Message Filtering*) in commands such as:

```
if "$h_subject:" contains "Make money fast" then ...
```

but there are also situations where it can be useful to refer to them in driver configurations.

If there is more than one header line with the same name (common with *Received:* headers), they are concatenated and inserted together, including their terminating newlines. However, once the string that is being built by concatenation exceeds 64 KB in length, further headers of the same name are ignored.

Operations on Substrings

A number of expansion items perform some operation on a portion of the expansion string, having first expanded it in its own right. They are all of the form:

```
${operator-name:substring}
```

In some cases, the operator name is followed by one or more argument values, separated by underscores. The substring starts immediately after the colon, and may have significant leading and/or trailing whitespace. In a real configuration, it always contains at least one expansion item; there is little point in writing a literal string to be operated on, because you could do the operation yourself and write the result instead.

Extracting the Initial Part of a Substring

Consider this common setting of the `file` option in an **appendfile** transport:

```
file = /var/mail/$local_part
```

On a system with very many mailboxes, it is desirable not to keep them all in the same directory, but rather to split them among several directories. A simple scheme for doing this would be:

```
file = /var/mail/${length_1:$local_part}/$local_part
```

which interposes another directory level. The name of the intermediate directory is computed from the local part by means of the `length` expansion operator, which extracts an initial substring from its argument, having first expanded it. Suppose the local part is *caesar*. On encountering the following:

```
${length_1:$local_part}
```

during the expansion process, Exim first expands the operator's argument, converting the item to:

```
${length_1:caesar}
```

The `length_1` operator then extracts the first character of `caesar`, before carrying on with the rest of the string, so that the final mailbox name becomes:

```
/var/mail/c/caesar
```

Assuming all local parts start with a letter, this distributes the user mailboxes among 26 subdirectories. If this were not enough, more characters from the local part could be used:

```
file = /var/mail/${length_2:$local_part}/$local_part
```

This in theory produces 676 (26×26) subdirectories. The problem with this approach is the letters in usernames are not usually spread evenly over the alphabet, and so some subdirectories are more heavily used than others. (See the section "Hashing Operators," later in this chapter, for a better way of handling this.)

Extracting an Arbitrary Part of a Substring

In addition to `length`, Exim has a `substr` operator that can be used to extract arbitrary substrings. For example:

```
${substr_3_2:$local_part}
```

extracts two characters starting at offset 3 in the local part. The first character in the string has offset zero. Suppose you want to include the name of the month in a filename for some reason. The variable $tod_full contains the current date and time in the form:

```
Fri, 07 Apr 2000 14:25:48 +0100
```

from which the month can be extracted by:

```
${substr_8_3:$tod_full}
```

If the starting offset of `substr` is greater than the string length, the result is an empty string; if the length plus starting offset is greater than the string length, the result is the righthand part of the string, starting from the offset.

The `substr` operator can take negative offset values to count from the righthand end of its operand. The last character is offset -1, the second-to-last is offset -2, and so on. For example:

```
${substr_-5_2:1234567}
```

yields 34. If the absolute value of a negative offset is greater than the length of the string, the substring starts at the beginning of the string, and the length is reduced by the amount of overshoot. For example:

```
${substr_-5_2:12}
```

yields an empty string, but:

```
${substr_-3_2:12}
```

yields 1. If the second number is omitted from `substr`, the remainder of the string is taken if the offset is positive. If it is negative, all characters in the string preceding the offset point are taken. For example:

```
${substr_4:penguin}    yields  uin
${substr_-4:penguin}   yields  pen
```

That is, an offset of -*n* with no length yields all but the last *n* characters of the substring.

Hashing Operators

For historical reasons, Exim has two hashing functions, one of which produces a string consisting of letters and digits, whereas the other produces one or two strings that are numbers. A setting for the one-letter mailbox subdirectory discussed earlier gives a more even spread:

```
file = /var/mail/${hash_1:$local_part}/$local_part
```

The `hash_1` operator applies a hashing algorithm that produces a hash string of length 1, chosen from the set of upper- and lowercase letters and digits. The subdirectory name is now a single character, out of 62 possibilities. Longer hash strings can be requested by giving different numbers after `hash`, and the character set from which they are chosen can be controlled by a second parameter.

However, the newer, numeric hashing function (`nhash`) gives a more even spread of results and can handle larger numbers of possibilities. Using the numeric hash, the setting:

```
file = /var/mail/${nhash_62:$local_part}/$local_part
```

achieves a similar effect to the earlier string hash, though now the names of the subdirectories are numerical strings in the range 0–61. For use on very large systems, `nhash` can be requested to produce two numbers computed from a single hash of the string. For example:

```
file = /var/mail/${nhash_8_64:$local_part}
```

When this is done, the two numbers are separated by a slash, so *caesar*'s mailbox might now be:

```
/var/mail/7/49/caesar
```

Forcing the Case of Letters

Two other operators that are sometimes useful are `uc` and `lc`, which force their arguments to upper- or lowercase, respectively. Local parts of addresses are, in general, case-sensitive according to the RFCs. Exim by default forces them to lowercase if they are in a local domain, because usernames on Unix systems are normally in lowercase.* However, it must preserve the case of local parts in remote addresses.

As an example, suppose a user wants to use a filter to file incoming messages in different mail folders for different senders, in a case-independent manner. A filter command such as this could be used:

```
save $home/mail/${lc:$sender_address}
```

The `lc` operator ensures that the entire sender address is in lowercase when it is used in the filename.

Character Translation

The `tr` expansion item translates single characters in strings into different characters, according to its arguments. The first argument is the string to be translated, and this is followed by two translation strings, normally of the same length. For example, the following is used to translate a comma-separated list into a colon-separated list:

```
${tr {a,b,c}{,}{:}}  yields  a:b:c
```

The second and third strings can contain more than one character. They correspond one-to-one, and each character in the second string that is found in the first string is replaced by the corresponding character in the third string. If there are duplicates in the second string, the last occurrence is used. If the third string is shorter than the second, its last character is replicated. However, if it is empty, no translation takes place.

* See the section "Handling Local Parts in a Case-Sensitive Manner," in Chapter 5, *Extending the Delivery Configuration*, for how to handle systems with mixed-case logins.

Text Substitution

A general string substitution facility is available. It is called `sg` because it operates like *sed* and Perl's `s` operator with the `/g` (global) option. It takes three arguments: the subject string, a regular expression, and a replacement string. For example:

```
${sg {abcdefabcdef}{abc}{**}}  yields  **def**def
```

The pattern is matched against the subject string, and the matched portion is replaced by the replacement string. A new match is then begun on the remainder of the subject, and this continues until the end of the subject is reached. If there are no matches, the yield is the unaltered subject string. Within the replacement string, the numeric variables $1, $2, and so on can be used to refer to captured substrings in the regular expression match. Because all three arguments are expanded before use, any `$` characters that are required in the regular expression or in the replacement string have to be escaped. For example:

```
${sg {abcdef}{^(...)(...)\$}{\$2\$1}}  yields  defabc
```

Conditional Expansion

A great deal of the power of the expansion mechanism comes from its ability to vary the results of expansion items depending on certain conditions. The basic conditional expansion item takes the following form:

```
${if condition {string1}{string2}}
```

The condition is tested, and if it is true, ***string1*** is expanded and used as the replacement for the whole item; if not, ***string2*** is used instead. A number of different conditions are available.

Testing for a Specific String

To test the exact value of a string, the `eq` condition is used. Here is an easy way to implement an exceptional mailbox location for just one user:

```
file = ${if eq{$local_part}{john}\
       {/home/john/inbox}\
       {/var/mail/$local_part}}
```

The condition that is tested is `eq{$local_part}{john}`; following the condition name `eq`, two substrings are given in curly brackets. Each is separately expanded and then they are compared for equality. If they are equal, the first of the following substrings is used; otherwise the second is used. So in this example, if the local part is *john*, the mailbox is */home/john/inbox*, whereas for all other local parts it is the result of expanding `/var/mail/$local_part`.

It is often useful to lay out conditions and other complicated expansion strings over several lines like this, because it makes them easier to read. Because the substrings are independently expanded, they may contain their own conditional expansions, which can make for very unreadable text. For example, if you want an exceptional mailbox location for two users, it could be done by setting:

```
file = ${if eq{$local_part}{john}\
       {/home/john/inbox}\
       {\
         ${if eq{$local_part}{jack}\
         {/home/jack/inbox}\
         {/var/mail/$local_part}}\
       }}
```

Laid out like this it is fairly easy to understand, compared with:

```
file = ${if eq{$local_part}{john\
       }{/home/john/inbox}{${if eq\
       {$local_part}{jack}{/home/jack/inbox}{\
       /var/mail/$local_part}}}}
```

Clearly, this approach to this particular requirement is usable only for a few cases, because it does not scale very well. For large numbers of exceptions, another technique should be used. For local parts that match a pattern, you could use a regular expression. Otherwise a file lookup could be used.

Negated Conditions

A separate condition for testing inequality is not provided because there is a general negation mechanism for conditions. Any condition preceded by an exclamation mark is negated, like this:

```
${if ! eq{$local_part}{john}{...
```

Of course, in the case of `eq`, another way of achieving the same effect is to transpose the order of the two strings that follow the condition.

Regular Expression Matching

If you want to test a string for something other than simple equality, you can use a regular expression. The `match` condition, like `eq`, takes the next two brace-enclosed substrings as its arguments. The second is interpreted as a regular expression and is matched against the first. For example:

```
${if match {$local_part}{^x\\d\\d}{...
```

tests whether a local part begins with `x` followed by at least two digits. The backslashes that form part of the regular expression have been doubled because they are treated as escape characters by the expander. If there are any dollar or brace characters in a regular expression, they too must be escaped. If the regular expression contains capturing parentheses, the captured substrings are available in the variables $1, $2, and so on during the expansion of the "success" string.* For example:

```
${if match {$local_part}{^x(\\d\\d)}{$1}}
```

not only tests $local_part as before, but yields the value of the two digits as its result via the expansion of $1. If there is no match, its yield is the empty string. When tests of this kind are nested, the values of the numeric variables ($1, $2, and so on) are remembered at the start of processing an `if` item, and restored afterwards.

Encrypted String Comparison

There is one more condition for testing a string, with a very specific purpose. When an Exim server is authenticating an SMTP connection† it may need to compare a cleartext password with one that has been encrypted (for example, a user password from */etc/passwd* or equivalent). This is done by encrypting the cleartext and comparing the result with the encrypted value Exim already has. It might seem that all that is needed is an operator to encrypt a string, but because of the way Unix passwords are encrypted, that is not sufficient. An encrypted password is stored with a two-character "salt" string, and the same value is needed in order to encrypt the string to be tested in the same way. The stored string contains both the salt and the result of the encryption.‡ To save having to quote it twice (once to encrypt and once to compare), there is a single condition called `crypteq` that does both jobs at once. Its first argument is the cleartext, and its second is the encrypted text, like this:

```
${if crypteq {mysecret}{ksUCNd4Cs61SI}{...
```

In this example, the cleartext `mysecret` is first encrypted by the `crypt()` function, using the salt `ks`, and the result is then compared with `UCNd4Cs61SI`. In most real cases, of course, the encrypted value is not included directly in the string like this. The second argument for `crypteq` is more likely to be a subexpansion that looks up a value in a file or database.

Forms of encryption other than that used by the `crypt()` function are used in some installations. LDAP has introduced a notation whereby an encrypted string is preceded by a string in braces that states how it was encrypted. Fortunately, an opening brace character is not valid as a "salt" character. If the encrypted string

* See Appendix B, *Regular Expressions*, for a reference description of regular expressions and captured substrings.

† For details of SMTP authentication, see Chapter 15, *Authentication, Encryption, and Other SMTP Processing*.

‡ For details of password encryption, see the specification of the `crypt()` function.

does not begin with a brace, encryption by `crypt()` is assumed by the `crypteq` condition. Otherwise, the contents of the braces must either be `crypt` (which has the same effect) or `md5`, for example:

```
{md5}CY9rzUYh03PK3k6DJie09g==
```

This indicates that the encrypted string is an MD5 hash of the cleartext. If you include such a string directly into an expanded string, you will have to quote the braces using backslashes, because they will otherwise be taken as part of the expansion syntax. For example:

```
${if crypteq{test}{\{md5\}CY9rzUYh03PK3k6DJie09g==}{1}{0}}
```

The `crypteq` condition is automatically included in the Exim binary when it is built with SMTP authentication support, but otherwise it has to be specially requested at build time.

PAM Authentication

Another expansion condition concerned with authentication is `pam`, which provides an interface to the *PAM* library that is available on some operating systems. PAM stands for *Pluggable Authentication Modules*, and it provides a framework for supporting different methods of authentication. The caller of PAM supplies a *service name* and an initial string that identifies a user. The authentication function then requests zero or more data strings from the caller, and it uses these as input to its authentication logic. In the most common case, a single data string is requested, which is in effect a password.

The `pam` expansion condition has a single argument that consists of a colon-separated list of strings. PAM is called with the service name `exim` and the first of the strings as the username. The remaining strings are passed to PAM in response to its data requests. The condition is true if PAM authenticates successfully. For example:

```
${if pam{mylogin:mypassword}{yes}{no}}
```

yields `yes` if the user *mylogin* successfully authenticates with the single data string `mypassword`, and `no` otherwise. The data for `pam` is rarely a fixed string like this; usually it is made up of variables containing values from the SMTP *AUTH* command (see Chapter 15).

If passwords that are to be used with PAM contain colon characters, the example just given will not work, because a colon in the password is interpreted as a delimiter in the list of strings to pass to PAM. This problem can be avoided by doubling any colons that may be present. If the password is in $2 for example, this could be used:

```
${if pam{mylogin:${sg{$2}{:}{::}}{yes}{no}}
```

In some operating environments, PAM authentication can be used only by a root process. When Exim is receiving incoming mail from remote hosts, it runs as the Exim user (provided one is defined), which sometimes causes problems with PAM in these environments. There is as yet no easy resolution of this problem.

Numeric Comparisons

Exim provides a number of conditions that test numeric values. These use familiar symbols such as `>` and `=`. They are written in a prefix notation, with the condition first, followed by two substrings that must (after their own subexpansion) take the form of optionally signed decimal integers. They are alternatively followed by one of the letters `K` or `M` (in either upper- or lowercase), signifying multiplication by 1024 or 1024×1024, respectively. For example:

```
${if > {$message_size}{10M}{...
```

tests whether the size of the message (which is contained in the $message_size variable) is greater than 10 MB. The available numeric conditions are:

`=`	equal
`==`	equal
`>`	greater
`>=`	greater or equal
`<`	less
`<=`	less or equal

The general negation facility provides for inequality testing.

Empty Variables and Nonexistent Header Lines

The `def` condition tests whether a variable contains any value, or whether a particular header line exists in a message. In the first case, it is followed by a colon and the variable name, and it is true only if the variable is not empty. For example:

```
${if def:sender_host_address {remote}{local}}
```

yields the string `remote` if $sender_host_address is not empty (indicating that the message did not originate on the local host), and `local` otherwise. Note that the variable name is given in the condition without a leading `$` character. If the variable does not exist, an error occurs.

In its second guise, `def` is followed by a colon, the string `header_` or `h_`, and the name of a message header line terminated by another colon, for example:

```
${if def:header_reply-to:{$h_reply-to:}{$h_from:}}
```

The condition tests for the existence of the header line in the message being processed, so in this case the yield of the expansion is the contents of *Reply-To:* if it exists; otherwise the yield is the contents of *From:*. The expander does not check whether there are actually any data characters in the header line or not.

File Existence

There are times when you may want to check on the existence of a file or directory, and adopt different strategies depending on whether it exists or not. The `exists` condition has a single string as an argument. It calls the `stat()` function to see if the string exists as a pathname in the filesystem, and is true if the function succeeds. Suppose you are moving user mailboxes from one directory to another. If a mailbox exists in the old directory, you want Exim to use it; otherwise you want to use or create a mailbox in the new directory. A setting of the `file` option like this could be used:

```
file = /var/\
  ${if exists{/var/oldmail/$local_part}{old}{new}}\
  mail/$local_part
```

For the local part *sue*, this expands to */var/oldmail/sue* if that file exists; otherwise to */var/newmail/sue*.

The State of a Message's Delivery

There are two conditions that have no data associated with them: `first_delivery` is true during the first delivery attempt on a message, but is false during any subsequent delivery attempts; `queue_running` is true during delivery attempts that have been started by a queue runner process, but is false otherwise. These allow you to adopt different directing and routing strategies at different times, if you want to. If you want a director or router to be used only at the first delivery attempt, you can set an option like this:

```
condition = ${if first_delivery {yes}{no}}
```

During the first delivery, the condition is true and the string expands to `yes`, which the `condition` option interprets as an instruction to run the director or router. In subsequent delivery attempts, the result is `no`, and so the director or router is skipped.

Combining Expansion Conditions

Several conditions can be combined into a single condition using the logical operators `and` and `or`. Each of these is followed by any number of conditions, each enclosed in braces. The entire list is also enclosed in braces, to show where it ends. This can make for some very unreadable text unless one is careful. The `and` condition is true only if all of its subconditions are true, while the `or` condition is true if any one subcondition is true. Here is a fanciful example that tests whether today is a leap day, spread over several lines for readability:

```
${if and
   {
   {eq {${substr_5_2:$tod_log}}{02}}
   {eq {${substr_8_2:$tod_log}}{29}}
   }
   {Today's the day}{Not today}}
```

The $tod_log variable contains the current date and time in this format:

```
2000-02-29 14:42:00
```

so the first condition tests the two digits for the month, and the second one tests the day number. The subconditions are tested from left to right, and only as many as necessary to establish the overall condition are fully evaluated. Subconditions may themselves use `and` and `or` if necessary.

Forcing Expansion Failure

We defined the conditional expansion item as a condition followed by two substrings: a "true" string and a "false" string. If the second string is empty (that is, if it would be coded as `{}`), it can be omitted. Sometimes, however, you don't want to use an empty string if the condition fails; you want to take more drastic action. Instead of a second string, you can supply the word `fail`, not in braces. This causes the entire expansion to fail, but in a way that the calling code can detect. We say "the expansion is forced to fail."

The result of this kind of failure depends on what the expanded string is being used for. In some cases it is no different from a failure caused by a syntax error, but in a number of other cases it causes whatever is being done to be skipped. For example, you can cause headers to be added to a message by setting the `headers_add` option on a director, router, or transport, containing a string to be added to the header section of a message. If you used something such as this:

```
headers_add = ${if eq {$sender_host_address}{}\
   {X-Postmaster: <postmaster@example.com>}\
   fail}
```

Exim would add an *X-Postmaster:* header to any messages that had been received locally (host address empty). When $sender_host_address is not empty, `fail` causes the string expansion to fail, which in this particular circumstance causes the addition of headers to be cancelled. Whenever `fail` in an expansion has a special effect like this, it is documented along with the option to which it applies.

Lookups in Expansion Strings

Another powerful feature of expansion strings is the ability to call Exim's lookup functions while expanding a string. This means that a portion of the string can be replaced by data obtained from a file or database, or the existence of a particular key in a file or database can be used to influence the result of the expansion.

The `lookup` expansion item is a form of conditional expansion, containing two substrings following the specification of the lookup. If the lookup succeeds, the first substring is expanded and used; if the lookup does not find any data, the second substring is used instead. Just as in the case of an `if` condition, the second substring may be absent, or the word `fail` may be used, as described in the previous section.

There are two different formats for the `lookup` item, depending on whether a single-key or query-style lookup is being used. We will introduce them by extending an example that was used earlier.

Single-Key Lookups in Expansion Strings

Recall that the following setting:

```
file = ${if eq{$local_part}{john}\
       {/home/john/inbox}\
       {/var/mail/$local_part}}
```

provides an easy way of specifying an exceptional location for the mailbox of just one user. With any more than a handful of exceptions, this technique does not work well. Instead, it would be better to create an indexed file containing the locations of all the special mailboxes. Laid out as a flat file, it might contain lines such as this:

```
john:      /home/john/inbox
jill:      /home/jill/inbox
alex:      /home/alex/mail/inbox
```

This file could be used directly in an **appendfile** transport:

```
file = ${lookup {$local_part} lsearch {/the/file} \
       {$value}{/var/mail/$local_part}}
```

This is an example of a `lookup` item for a single-key search type. The item is introduced by `${lookup`; this is followed by a substring in braces that defines the key to be looked up, after it has been separately expanded. In this case, it is the local part. Then there is a word that specifies the type of lookup, in this case `lsearch`, followed by a substring containing data for the search. For a single-key search type, the data is the name of the file to be searched.

The remainder of the item consists of two braced substrings. If the lookup succeeds, the first string is expanded and used; during its expansion, the variable $value contains the data that was looked up. If the lookup fails, the second string (if present) is expanded and used. In our example, if the local part is *jill*, the lookup succeeds, and $value contains `/home/jill/inbox` during the expansion of the first string. As this consists of `$value` only, the result of the entire lookup is `/home/jill/inbox`. If a local part that is not in the file is looked up, no data is found, and so the second string, `/var/mail/$local_part`, is expanded and used.

Partial Lookups in Expansion Strings

Partial lookups, described in the section "Partial Matching in Single-Key Lookups," in Chapter 16, *File and Database Lookups*, can be used for single-key lookup types, and when a partial lookup succeeds, the variables $1 and $2 contain the wild and nonwild parts of the key during the expansion of the replacement substring. They return to their earlier values at the end of the lookup item. Consider a file of domain names containing:

```
one.example
*.two.example
```

and a lookup item of the form:

```
${lookup {$domain} partial-lsearch \
  {/the/file}{wild="$1" notwild="$2"}}
```

When $domain contains `one.example`, the result is:

```
wild="" notwild="one.example"
```

because no partial matching was done. However, if $domain contains `twenty.two.example`, the result is:

```
wild="twenty" notwild="two.example"
```

Single-Key Lookup Failures

If Exim cannot attempt the lookup because of some problem (for example, if it cannot open the file), the entire string expansion fails, in a similar way to a syntax error. You can guard against the particular case of a nonexistent file by using the exists test described earlier. For example:

```
file = ${if exists{/the/file}\
        {\
        ${lookup {$local_part} lsearch {/the/file} \
        {$value}\
        {/var/mail/$local_part}}\
        }\
        {/var/mail/$local_part}}
```

Query-Style Lookups in Expansion Strings

A large linear file should be turned into an indexed structure of some kind (DBM or cdb) to improve performance, or the data could be stored in one of the databases Exim supports. Here is the same configuration setting, but this time it's looking up the mailbox using MySQL:

```
file = ${lookup mysql \
       {select mbox from users where \
       id='${quote_mysql:$local_part}'} \
       {$value} \
       {/var/mail/$local_part}}
```

This example assumes there is a table called `users` with fields called `id` and `mbox`. When a query-style lookup is used in an expansion string, you do not specify a separate key. Instead, the lookup type name follows `${lookup` immediately and is itself followed by a substring that forms the database query. The success and failure substrings then follow as before.

Reducing the Number of Database Queries

On a busy system, a lookup such as the previous example, which requires a database call for every delivery, might result in poor performance. An improvement could be obtained by dumping the data from the database every night, and building (for example) a cdb file that would give much better performance. However, additions to the database would not then take immediate effect. The best of both worlds could be achieved by checking the cdb file first, and doing a database lookup only if that failed. For example:

```
file = \
  ${lookup {$local_part} cdb {/the/cdb/file} \
    {$value}\
      {\
      ${lookup mysql \
      {select mbox from users where \
      id='${quote_mysql:$local_part}'} \
{$value} \
      {/var/mail/$local_part}}\
      }\
    }
```

There are some further remarks on the topic of reducing the number of database queries in the section "Extracting Named Fields from a Line of Data," later in this chapter.

Defaults for Lookups in Expansion Strings

When using single-key lookups in expansion strings, the lookup type name may be followed by `*` or `*@` to request default lookups, as described in the section "Default Values in Single-Key Lookups," in Chapter 16. Alternatively, the "false" substring can contain an explicit second lookup to be done if the first one fails, and this works for both single-key and query-style lookups.

A Shorthand for One Common Case

Sometimes, you know a lookup will succeed because of some previous test. For example, if a set of domains is defined by a lookup, you can assume that reusing the same lookup to obtain data for one of those domains is not going to fail. In the section "Virtual Domains," in Chapter 5, we show this director for virtual domains:

```
virtuals:
  driver = aliasfile
  domains = cdb;/etc/virtuals
  file = /etc/$domain.aliases
  search_type = lsearch
  no_more
```

Suppose that instead of a fixed pattern for each domain's alias file, you want to look up each name individually, storing the data in */etc/virtuals*. The director becomes:

```
virtuals:
  driver = aliasfile
  domains = cdb;/etc/virtuals
  file = ${lookup{$domain}cdb{/etc/virtuals}{$value}}
  search_type = lsearch
  no_more
```

There is no need to specify a "false" substring for the expansion lookup, because it will never be needed. For this common case, as long as you are using Exim Release 3.20 or later, you can omit the "true" substring as well, and specify just this:

```
file = ${lookup{$domain}cdb{/etc/virtuals}}
```

When there are no substrings following a lookup specification, Exim assumes that `{$value}` is wanted.

Extracting Fields from Substrings

There are three expansion items that extract data fields from substrings, having first expanded them.

Splitting Up Addresses

`local_part` and `domain` are operators that extract the local part and the domain from an address, respectively. For example, if the sender of a message were *brutus@rome.example.com*, then:

```
local part is ${local_part:$sender_address}
domain is ${domain:$sender_address}
```

would expand to:

```
local part is brutus
domain is rome.example.com
```

Extracting Named Fields from a Line of Data

Suppose you want to pick out individual field values from a data file containing lines such as this:

```
trajan: uid=142 gid=241 home=/homes/trajan
```

An `lsearch` lookup on the key `trajan` returns the rest of the line of data; the `extract` expansion item can be used to split it up. For example, to obtain the value of the `uid` field:

```
${extract{uid}{${lookup{trajan}lsearch{/the/file}{$value}fail}}}
```

There are two substrings that follow `extract`; the first is a name, and the second is a string of *`name`*`=`*`value`* items from which it extracts the value that matches the given name. In this example, the string is obtained by a lookup, which is the most common case. The lookup of the key `trajan` yields:

```
uid=142 gid=241 home=/homes/trajan
```

so the `extract` item becomes:

```
${extract{uid}{uid=142 gid=241 home=/homes/trajan}}
```

before the extraction operation is applied. If any of the values in the data contain whitespace, they must be enclosed in double quotes, and within double quotes, normal escape processing takes place.*

If the name is not found in the data string, the item is replaced by the empty string. However, from Exim Release 3.20 onwards, the `extract` item can also be used in a similar way to a lookup or conditional item. If two further argument strings are given, the first is used when the extraction succeeds, with $value containing the extracted substring, and the second substring is used when the extraction fails. Once again, "fail" can be specified instead of a second substring to force expansion failure. For example:

```
${extract{uid}{uid=142 gid=241 home=/homes/trajan}{userid=$value}}
```

yields `userid=142`.

If you are using query-style lookups, you can often select individual fields within the query language. However, if a number of fields are required, it is better to read them from the database together and use `extract` to separate them, because the results of database lookups are cached. Consider the following two settings, which might appear on a local transport's configuration, to specify the uid and gid under which it is to run:

```
user  = ${lookup mysql \
  {select uid from accounts where \
  id='${quote_mysql:$local_part}'}\
  {$value}}
group = ${lookup mysql \
  {select gid from accounts where \
  id='${quote_mysql:$local_part}'}\
  {$value}}
```

Two separate database queries are required, one for each option. If instead these settings are used:

```
user  = ${extract{uid}{\
        ${lookup mysql \
        {select uid,gid from accounts where \
          id='${quote_mysql:$local_part}'}{$value}}\
        }}
group = ${extract{gid}{\
        ${lookup mysql \
        {select uid,gid from accounts where \
          id='${quote_mysql:$local_part}'}{$value}}\
        }}
```

* \n is turned into newline, for example, and backslash must appear as \\.

only a single database call actually occurs, because the two queries are identical, so the cached value is used for the second one. Exim caches only the most recent query, but this is sufficient to give substantial benefit.

Extracting Unnamed Fields from a Line of Data

There is a second form of the `extract` expansion item that can be used on data strings that are separated by particular characters. A good example is the password file, */etc/passwd*, where the fields are separated by colons. To extract a user's real name from this file, an expansion such as this can be used:

```
${extract{4}{:}\
  {${lookup{hadrian}lsearch{/etc/passwd}{$value}fail}}}
```

If the line in */etc/passwd* is:

```
hadrian:x:42:99:Hadrian IV::/bin/bash
```

the lookup yields:

```
x:42:99:Hadrian IV::/bin/bash
```

and the result of the extraction is the fourth colon-separated field in this data, namely, `Hadrian IV`.

This form of `extract` is distinguished from the other by having a first argument consisting entirely of digits. The second argument is a list of field separator characters, any one of which can be used in the data. In other words, the separators are always a single character, not a string. Two successive separators mean that the field between them is empty (the fifth field in the previous example). If the field number in the expansion is zero, the entire string is returned; if it is greater than the number of fields, the empty string is returned.

However, from Exim Release 3.20, as in the case of the other form of `extract`, you can optionally supply two additional substrings that are used in a similar way to the substrings in a lookup or conditional item. The first is used if the field is found, with $value containing the extracted data, and the second substring is used otherwise. Once again, "fail" can be specified instead of a second substring, to force expansion failure.

IP Address Masking

An IP network is defined using an IP address and a mask. For example, 192.168.34.192/26 defines the network consisting of all IP addresses whose most significant 26 bits are the same as those of 192.168.34.192. To check whether a given host is in this network, it is necessary to mask the least significant bits of its IP address (that is, convert them into zeros) before comparing it with the network address. There are some built-in host tests that automatically take care of masking, but in order to let you write custom tests, a masking expansion operator exists. When processing a message that came from the host 192.168.34.199, the string:

```
${mask:$sender_host_address/26}
```

would expand to the string `192.168.34.192/26`. This could be compared against a fixed value, or looked up in a file. This poses a small problem in the case of IPv6 addresses, which are normally written using colons to separate the components, because a colon is the key terminator in data files that are in `lsearch` format (that is, in the same form as alias files). A normal IPv6 address could not therefore be a key in such a file. In order to make this possible, the `mask` operator outputs IPv6 addresses using dots as separators instead of colons. For example, the string:

```
${mask:5f03:1200:836f:0a00:000a:0800:200a:c031/99}
```

expands to:

```
5f03.1200.836f.0a00.000a.0800.2000.0000/99
```

which could be used as an `lsearch` key. Letters in IPv6 addresses are always output in lowercase by the `mask` operator.

Quoting

When data from a message is included in an expansion string by a variable or header insertion, problems can occur if the inserted data contains unanticipated characters. Local parts can contain all manner of special characters if they are correctly quoted using RFC 822's rules. The following addresses are all valid:

```
O'Reilly@ora.com.example
double\"quote@weird.example
"two words"@weird.example
"abc@pqr"@some.domain.example
abc\@pqr@some.domain.example
```

When Exim receives an address, it strips out the RFC 822 quoting so as to obtain a "canonical" representation of the local part. The last two examples have the same canonical local parts. Suppose that $local_part was being used in a MySQL lookup query containing the fragment:

```
 ... where id='$local_part' ...
```

then the first example would break it. Such problems can be avoided by using the appropriate quoting mechanism in the string expansion. Several are provided, for use in different circumstances.

Quoting Addresses

The `quote` operator puts its argument into double quotes if it contains anything other than letters, digits, underscores, dots, or hyphens. Any occurrences of double quotes and backslashes are escaped with a backslash. This kind of quoting is useful if a new mail address is being created from an old one for some reason. For example, suppose you wanted to send unknown local parts in your local domain to some other domain, retaining the same local part, but changing the domain. You could do this using a **smartuser** director such as this as your final director:

```
send_elsewhere:
  driver = smartuser
  new_address = $local_part@dead-letter.example.com
```

As this is the last director, local parts that the other directors could not handle would be passed to it. The example would work fine until somebody sent a message with a local part containing an @ character, for example:

```
"malicious@example"@your.domain
```

whereupon Exim would report a syntax error in the new address, because the value of $local_part would be `malicious@example` after removing the RFC 822 quoting. To guard against this, the option is better defined as:

```
new_address = ${quote:$local_part}@dead-letter.example.com
```

which would cause the generated local part to be quoted if necessary.

Quoting Data for Regular Expressions

The `rxquote` operator inserts backslashes before any nonalphanumeric characters in its argument. As its name suggests, it is used when inserting data that is to be interpreted literally into regular expressions. If you wanted to check whether the current delivery address was mentioned in the *To:* header of a message, you could write an expansion conditional such as this:

```
${if match{$h_to:}{^.*$local_part@$domain} {...
```

However, any regular expression metacharacters in the local part or the domain would break the regular expression, and as a dot is such a character, the domain is almost certain to cause trouble. The correct way to write this is:

```
${if match{$h_to:}{^.*${rxquote:$local_part@$domain}} {...
```

Quoting Data in Lookup Queries

There are special quoting operators for each of the query-style lookup types:

NIS+

The effect of the `quote_nisplus` expansion operator is to double any quote characters within the text.

LDAP

Two levels of quoting are required in LDAP queries, the first for LDAP itself, and the second because the LDAP query is represented as a URL. The `quote_ldap` expansion operator implements the following rules:

- For LDAP quoting, the characters `#,+"\<>;*()` have to be preceded by a backslash.*
- For URL quoting, all characters except alphanumerics and `!$'()*+-._` are replaced by `%`*xx*, where *xx* is the hexadecimal character code. Note that backslash has to be quoted in a URL, so characters that are escaped for LDAP end up preceded by `%5C` in the final encoding.

MySQL

The `quote_mysql` expansion operator converts newline, tab, carriage return, and backspace to `\n`, `\t`, `\r`, and `\b`, respectively, and the characters `'"\` are escaped with backslashes. Percent and underscore are special only in contexts where they can be wildcards, and MySQL does not permit them to be quoted elsewhere, so they are not affected by the `quote_mysql` operator.

PostgreSQL

The `quote_pgsql` expansion operator converts newline, tab, carriage return, and backspace to `\n`, `\t`, `\r`, and `\b`, respectively, and the characters `'"\` are escaped with backslashes, as for `mysql_quote`. However, percent and underscore are treated differently. PostgreSQL allows them to be quoted in contexts where they are not special. For example, an SQL fragment such as:

```
where id="ab\%cd"
```

has the same effect as:

```
where id="ab%cd"
```

which is not the case for MySQL. Percent and underscore are therefore escaped by `quote_pgsql`.

* In fact, only some of these need be quoted in Distinguished Names, and others in LDAP filters, but it does no harm to have a single quoting rule for all of them.

Quoting Printing Data

The final quoting operator is called `escape`, and it is for use when inserted data is required to contain printing characters only. It converts any nonprinting characters into escape sequences starting with a backslash.* For example, a newline character is turned into `\n`, and a backspace into `\010`. Local parts that are quoted in email addresses may contain such characters, though usually it is only persons bent on mischief that create them. Message header lines are another place where nonprinting characters may occur. As an example of where `escape` might be used, consider the creation of an automatic reply to an incoming message. The details of how to do this can be found in the section "The autoreply Transport," in Chapter 9, *The Transports*, but you don't need to know them to follow this example, which just shows how the *Subject:* line of the reply might be specified:

```
subject = Re: message from $sender_address to $local_part
```

That should not cause any delivery trouble, whatever the contents of the local part and sender address, but it could be very confusing if there were a lot of backspaces, for example, in one of the variables. A safer way to write this is:

```
subject = Re: message from ${escape:$sender_address} \
  to ${escape:$local_part}
```

Reexpansion

The `expand` operator first expands its argument substring like all the other operators, but then passes it through the expander for a second time. The most common use is when the first expansion does a lookup, because it allows the data that is looked up to contain expansion variables. Suppose the `file` option for local delivery is written like this:

```
file = ${lookup{$local_part}lsearch{/etc/mailboxes}\
       {${expand:$value}}{/var/mail/$local_part}}
```

To find the name of the mailbox file, the local part is looked up in */etc/mailboxes*. If no data is found, the string `/var/mail/$local_part` is expanded and used. Otherwise the value that was looked up is used, but it is first re-expanded. The file */etc/mailboxes* could contain lines such as this:

```
jim:   /home/jim/inbox
```

* Whether characters with the most significant bit set (so-called "8-bit" characters) count as printing or nonprinting is controlled by the `print_topbitchars` option.

for which the additional expansion would have no effect, but it could also contain lines like this:

```
    jon:   ${if eq{$h_precedence:}{bulk}{/dev/null}{/var/mail/jon}}
```

which discards messages with a *Precedence:* header line whose value is `bulk`, by causing them to be written to */dev/null*.*

Running Embedded Perl

The facilities available for string expansion allow for quite sophisticated transformations on strings. Nevertheless, there is always somebody who wants to do more. The ultimate sledgehammer is to run a Perl function as part of a string expansion. In order to do this, Exim has to be built with support for embedded Perl.† Access to Perl subroutines is via a configuration option called `perl_startup`, which defines a set of Perl subroutines, and the expansion string operator `${perl ...}`, which causes them to be run. If there is no `perl_startup` option in the Exim configuration file, no Perl interpreter is started, and there is almost no overhead for Exim (since none of the Perl library is paged in unless it is used).

If there is a `perl_startup` option, the associated value is taken to be Perl code, which is executed in a newly created Perl interpreter. It is not expanded in the Exim sense, so you do not need backslashes before any characters to escape special meanings. The option should usually be something such as:

```
    perl_startup = do '/etc/exim.pl'
```

where */etc/exim.pl* contains Perl code that defines the subroutines you want to use. Exim can be configured either to start up a Perl interpreter as soon as it starts to execute, or to wait until the first time the interpreter is needed. Starting the interpreter at the beginning ensures that it is done while Exim still has its setuid privilege, which might be needed to gain access to initialization files, but imposes an unnecessary overhead if Perl is not used in a particular run. By default, the interpreter is started only when it is needed, but this can be changed, as follows:

- Setting `perl_at_start` (a Boolean option) in the configuration requests a startup when Exim starts.
- The command-line option *-ps* also requests a startup when Exim starts, overriding a false setting of `perl_at_start`.

* This is not a recommended way of achieving this effect; it is just an example to demonstrate the `expand` operator.

† See Chapter 22, *Building and Installing Exim.*

- There is also a command-line option *-pd* (for delay) that suppresses the initial startup, even if `perl_at_start` is set.

When the configuration file includes a `perl_startup` option, string expansion items can call the Perl subroutines defined by the `perl_startup` code like this:

```
${perl{func}{argument1}{argument2} ... }
```

An item such as this calls the subroutine `func` with the given arguments (having first expanded the arguments). A minimum of zero and a maximum of eight arguments may be passed. Passing more than this results in an expansion failure. The return value of the subroutine is inserted into the expanded string, unless the return value is `undef`. In this case, the expansion fails in the same way as an explicit `fail` on an `if` or `lookup` item. If the subroutine aborts by obeying Perl's `die` function, the expansion fails with the error message that was passed to `die`.

The Perl interpreter is not run in a separate process, so when it is called from an expansion string, its uid and gid are those of the Exim process. In particular, initializing the interpreter when Exim starts to run causes it to run as root-only during its initialization; it does not cause subsequently called subroutines to run as root.

Within any Perl code called from Exim, the function `Exim::expand_string` is available to call back into Exim's string expander. This helps to reduce the number of arguments you need to pass to a Perl subroutine. For example, the Perl code:

```
my $lp = Exim::expand_string('$local_part');
```

makes the current value of $local_part available in the Perl variable $lp. Note the use of single quotes to protect against $local_part being interpolated as a Perl variable.

If the string expansion is forced to fail, the result of `Exim::expand_string` is `undef`. If there is a syntax error in the expansion string, the Perl call from Exim's expansion string fails with an appropriate error message, in the same way as if `die` were used.

Testing String Expansions

If you are setting up some complicated expansion string, perhaps involving lookups, conditionals, or regular expressions, it is helpful to be able to test it in isolation before you try it in an Exim configuration file. If Exim is called with the *-be* option:

```
exim -be
```

it doesn't perform any mail handling functions at all. Instead, if there are any command-line arguments, it expands each one and writes it as a separate line to the standard output. Otherwise, it reads lines from its standard input, expands them, and writes them to the standard output. It prompts for each line with an angle bracket. For example:

```
$ exim -be '$tod_log'
2000-02-10 15:51:00
$ exim -be
> ${lookup{root}lsearch{/etc/passwd}{$value}}
x:0:1:Super-User:/:/bin/sh
>
```

This facility allows you to test the general expansion functionality, but, because no message is being processed, you cannot make use of variables such as $local_part that relate to messages.

If you develop an expanded string in this way, and subsequently copy it into a configuration setting that is enclosed in quotes, remember that you will have to double any backslashes that it contains. There are not likely to be any unless you have used regular expressions, or needed to include literal dollar, backslash, or curly brackets in your string.

18

Domain, Host, and Address Lists

Lists of domains, hosts, or addresses are used in a number of Exim's options. In Chapter 3, *Exim Overview*, we introduced them with this example of a list of local domains:

```
local_domains = tiber.rivers.example:\
                *.cities.example:\
                dbm;/usr/exim/domains
```

This list contains three different kinds of item, but in fact these are not the only possibilities. In this chapter we will explore all the variations, not only for domain lists, but also for host and address lists.

Each list can be thought of as defining a set of domains, hosts, or addresses, respectively. When Exim is testing to see whether a domain (or host, or address) matches an item in a list, it asks the question "Is this domain (or host, or address) in the set defined by this list?" It scans the list from left to right, checking against each item in turn. As soon as an item matches, the scan stops.

For example, using the previous setting of `local_domains`, Exim first checks to see if the domain it is testing is *tiber.rivers.example*. If it is, the answer to the implied question is "Yes, it is a local domain." Otherwise, Exim goes on to check whether the domain ends with *.cities.example*, and if not, it looks up the domain in the file. If that fails, the answer is "No, this is not a local domain."

All lists use colons as separator characters by default, and whitespace at either end of an item is ignored. If you need to include a literal colon in an item, it must be doubled. Unfortunately, this is necessary for all colons that appear in IPv6 addresses. For example:

```
local_interfaces = 127.0.0.1: ::::1
```

contains two items: the IPv4 address `127.0.0.1` and the IPv6 address `::1`. The space after the first colon is vital; without it, the list would be incorrectly interpreted as the two items `127.0.0.1::` and `1`.

From Release 3.14 of Exim onward, it has been possible to change the separator character in lists, to make it easier to deal with IPv6 addresses. If a list starts with a < character that is immediately followed by a nonalphanumeric printing character (excluding whitespace), that character is used as the separator. The earlier example could be rewritten to use + as the separator, like this:

```
local_interfaces = <+ 127.0.0.1 + ::1
```

Negative Items in Lists

Sometimes it is useful to have exceptions to wildcard patterns. Suppose, for example, we wanted to exclude *athens.city.example* from the set of local domains, while retaining all others ending in *.cities.example*. The previous list items are *positive* items; that is, if they match, the answer is "yes." It is also possible to have *negative* items, which, if they match, cause the answer to be "no," and this provides exactly the feature we need. An exclamation mark preceding any item negates it.* Consider this example:

```
local_domains = !athens.cities.example:\
                *.cities.example
```

If the domain is *athens.cities.example*, the first item matches. Because it it negated, this causes the answer to be "no." Otherwise the domain is xmatched against **.cities.example*.

* There may be optional whitespace between the exclamation mark and the item.

If you are using a macro (see the section "Macros in the Configuration File," in Chapter 4, *Exim Operations Overview*) in your configuration file to define a number of items for a list, you cannot negate by putting an exclamation mark in front of the macro name because macros work by simple text substitution. For example, if you have defined:

```
LOCAL = domain1 : domain2
```

then:

```
domains = !LOCAL
```

is *not* the same as:

```
domains = !domain1 : !domain2
```

If the last item in a list is a negative item, it changes what happens if the end of the list is reached without anything matching. In the examples we've used so far, reaching the end of the list provokes a "no" answer. However, for a list such as this:

```
queue_remote_domains = !*.mydomain.example
```

reaching the end of the list causes a "yes" answer because that seems the natural interpretation of such a list. In effect, a list that ends with a negative item is treated as if it had an additional "matches everything" item at the end, so the previous example behaves exactly the same as:

```
queue_remote_domains = !*.mydomain.example : *
```

List Items in Files

In any of these lists, an item beginning with a slash is interpreted as the name of a file that contains out-of-line items, one per line. Empty lines in the file are ignored. The other lines are interpolated into the list exactly as if they appeared inline, except that the file is read afresh each time the list is scanned. A file may not, however, contain the names of other files. For domain and host lists, if a # character appears anywhere in a line of the file, it and all following characters on the line are ignored. For address lists, # must be followed by whitespace to be recognized as introducing a comment because it can legitimately form part of an address. If a plain filename is preceded by an exclamation mark, the sense of any match within the file is inverted. For example, with the setting:

```
hold_domains = !/etc/nohold-domains
```

and a file containing the lines:

```
!a.b.c
*.b.c
```

a.b.c is in the set of domains defined by `hold_domains`, whereas any domain matching **.b.c* is not.

Lookup Items in Lists

Lists may also contain lookup items; these operate differently for the three kinds of list, so they are described separately later. However, it is important to realize that a lookup, even if it uses the `lsearch` lookup method, is not the same as an interpolated file, as just described. The earlier example file could not be used for an `lsearch` lookup because the data in a lookup file is fixed: it cannot contain any negation or wildcarding.* Consequently, there is an important difference between (for example):

```
local_domains = /etc/local-mail-domains
```

and:

```
local_domains = lsearch;/etc/local-mail-domains
```

even though the file is read sequentially in both cases. In the first case, each line of the file is interpolated just as if it appeared inline, and may contain any item type other than a further filename. For example, it could contain items beginning with asterisks, or regular expressions. However, if a lookup is used (the second case), the keys in the file are not interpreted specially; they are always literal strings.

Domain Lists

Domain lists are used for specifying sets of mail domains for various purposes, such as which domains are local, or which are acceptable for relaying. The following types of items may appear in a domain list:

- If an item consists of a single @ character, it matches the local hostname, as set in the `primary_hostname` option. This makes it possible to use the same

* The partial and default features of single-key lookups are implemented by multiple probes of the file.

configuration file on several different hosts that differ only in their names. For example:

```
local_domains = @ : plc.com.example
```

ensures that the local hostname is a local domain.

- If an item starts with an asterisk, the remaining characters of the item are compared with the terminating characters of the domain. The use of an asterisk in domain lists differs from its use in partial matching lookups. In a domain list, the character following the asterisk need not be a dot, whereas partial matching works only in terms of dot-separated components. For example, a domain list such as:

  ```
  local_domains = *key.example
  ```

  matches *donkey.example* as well as *cipher.key.example*. Note that an asterisk may appear only at the start of an item to denote the commonly required "ends with" test. More complex wildcard matching requires the use of a regular expression (see the next bullet).

- If an item starts with a circumflex character, it is treated as a regular expression, and matched against the domain using a regular expression matching function. For example:

  ```
  local_domains = ^mta\d{3}\.plc\.example$
  ```

  specifies that any domain named *mta* followed by three digits and *.plc.example* is a local domain. The circumflex that introduces a regular expression is treated as part of the expression. This means that it "anchors" the expression to the start of the domain name but you can start with `^.*` to accommodate arbitrary leading characters if you need to. A description of the regular expressions that Exim supports can be found in Appendix B, *Regular Expressions*.

  There are some cases in which a domain list is the result of string expansion, for example the `domains` option in routers and directors. In these cases you must escape any backslash, dollar, and curly bracket (brace) characters in regular expressions to prevent them from being interpreted by the string expander.*

* If the string is specified in quotes, the resulting backslashes must themselves also be escaped.

- If an item starts with the name of a single-key lookup type followed by a semicolon (for example, `dbm;` or `lsearch;`), the remainder of the item must be a filename in a suitable format for the lookup type. For example, for `dbm;` it must be an absolute path:

  ```
  hold_domains = dbm;/etc/holddomains.db
  ```

  The appropriate type of lookup is done on the file using the domain name as the key. If the lookup succeeds, the domain that was looked up matches the item.

- Any of the single-key lookup type names may be preceded by `partial`*n*`-`, where the *n* is optional, for example:

  ```
  partial-dbm;/partial/domains
  ```

  This causes partial matching logic to be invoked; a description of how this works is given in the section "Partial Matching in Single-Key Lookups," in Chapter 16, *File and Database Lookups*.

- Any of the single-key lookup types may be followed by an asterisk. This causes a default lookup for a key consisting of a single asterisk to be done if the original lookup fails. This is not a useful feature when using a domain list to select particular domains (because any domain would match), but it might have value if the result of the lookup is being used via the $domain_data expansion variable (described later in this section).

- If the item starts with the name of a query-style lookup type followed by a semicolon (for example, `nisplus;` or `ldap;`), the remainder of the item must be an appropriate query for the lookup type. The query is expanded before use and the expansion variable $key can be used to insert the domain being tested into the query. For example:

  ```
  local_domains = \
    mysql; select domain from localdomains where domain='$key';
  ```

  If the lookup succeeds, the domain that is being tested matches the item. Note that this is not looking up a list of domains to test; it is testing a single domain by running a query (which normally would refer to the domain).

- If none of the earlier cases apply, a straight textual comparison is made between the item and the domain.

Lookups in domain lists are a way of obtaining a "yes" or "no" answer about a domain's membership of a particular set of domains. The data that is looked up as part of the test is normally discarded. However, there is one case when it is retained. The `domains` and `local_parts` options on a director or router check the domain and local part, respectively, before running the driver. If either is matched by means of a lookup, Exim sets the $domain_data and $local_part_data variables, respectively, to the data that was looked up, for the duration of the driver. These values can be used in other configuration options. From Exim Release 3.14, this data is also available to any transport that is run as a result of the director or router accepting the address.

Here is an (unrealistic) example of a domain list that uses several different kinds of items:

```
local_domains = \
  @:\
  lib.unseen.edu.example:\
  *.foundation.fict.example:\
  ^[1-2]\d{3}\.fict\.example$:\
  partial-dbm;/opt/penguin/example:\
  nis;domains.byname:\
  nisplus;[name=$key,status=local],domains.org_dir
```

There are some fairly obvious processing trade-offs among the various matching modes. Using an asterisk is faster than using a regular expression, and listing a few names explicitly probably is too. The use of a file or database lookup is expensive, but it may be the only option if hundreds of names are required. Because the items are tested in order, it makes sense to put the most commonly matched items earlier in the string.

Some domain lists, of which `local_domains` is the prime example, are scanned frequently by Exim. It is therefore important that any lookups they use be quick and unlikely to defer. You should not, for instance, set `local_domains` to a lookup on some heavily loaded database server running on a host on some remote network. Lookups should normally use a local file or a server running on the local host, or at least a host on the local LAN.

Host Lists

Host lists are used to control what remote hosts are allowed to do (for example, use the local host as a relay).

Host Checks by IP Address

When Exim receives an SMTP call from another host, the only reliable identification it initially has for the host is its IP address. This address is used for checking when any of these items is encountered in a host list:

- If the entire item is `*`, it matches any IP address, and therefore any host. For example:

  ```
  sender_verify_hosts = *
  ```

  specifies that, if `sender_verify` is turned on, it will apply to all hosts.

- If the item is an IP address, it is compared with the IP address of the subject host. For example:

  ```
  host_accept_relay = 10.8.43.23
  ```

  allows relaying from the host with that particular IP address. If an IPv4 host calls an IPv6 host, the incoming address actually appears in the IPv6 host as `::ffff:`*`v4address`*. When such an address is tested against a host list, it is converted into a traditional IPv4 address first.

- If the item is an IP address followed by a slash and a mask length, for example:

  ```
  host_lookup = 10.11.0.0/16
  ```

  it is matched against the IP address of the subject host under the given mask, which specifies the number of address bits that must match starting from the most significant end. Thus, an entire network of hosts can be included (or excluded) by a single item. IPv4 addresses are given in the normal "dotted-quad" notation. IPv6 addresses are given in colon-separated format, but the colons have to be doubled so as not to be taken as item separators. This example shows both kinds of addresses:

  ```
  receiver_unqualified_hosts = 192.168.0.0/12: \
                               5f03::1200::836f::::/48
  ```

  Colons in IPv6 addresses must be doubled only when such addresses appear inline in a host list. Doubling is not required (and must not be done) when IPv6 addresses appear in a file. For example:

  ```
  receiver_unqualified_hosts = /opt/exim/unqualnets
  ```

  could make use of a file containing:

  ```
  192.168.0.0/12
  5f03:1200:836f::/48
  ```

  to have exactly the same effect as the earlier example, though it is of course less efficient for a small number of addresses. If you are using Exim 3.14 or later, you can change the separator character in the list to avoid having to double colons in IPv6 addresses. For example:

  ```
  receiver_unqualified_hosts = <+ 192.168.0.0/12+ \
                                  5f03:1200:836f::/48
  ```

- If the item is of the form:

  ```
  netnumber-search-type;search-data
  ```

  for example:

  ```
  net24-dbm;/etc/networks.db
  ```

  the IP address of the subject host is masked using *number* as the mask length. A textual string is constructed from the masked value, followed by the mask, and this is used as the key for the lookup. For example, if the host's IP address is 192.168.34.6, the key that is looked up for the earlier example is `192.168.34.0/24`.

  IPv6 addresses are converted to a text value using lowercase letters and dots as separators instead of the more common colon; a colon is the key terminator in `lsearch` files, which prevents any string containing a colon from being used as a key. Full, unabbreviated IPv6 addresses are always used.

- If the item is of the form:

  ```
  net-search-type;search-data
  ```

  the text form of the IP address of the subject host is used unmasked as the lookup key string. This is not the same as specifying `net32` for an IPv4 address or `net128` for an IPv6 address, as the mask value is not included in the key. However, IPv6 addresses are still converted to an unabbreviated form using lowercase letters, with dots as separators.

Host Checking Using Forward Lookup

If an item in a host list is a plain domain name, for example:

```
host_accept_relay = my.friend.example
```

Exim calls `gethostbyname()` to find its IP addresses. This typically causes a forward DNS lookup of the name. In some cases other sources of information such as */etc/hosts* may be used though, depending on the way your operating system is set up. The result is compared with the IP address of the host being checked. The primary name of the local host can be included in a host list by an item consisting of just the character `@`; this makes it possible to use the same configuration on several hosts that differ only in their names.

Host Checking Using Reverse Lookup

The remaining types of item that can appear in host lists are wildcard, patterns for matching against the hostname. If this is not already known, Exim calls `gethostbyaddr()` to obtain it from the IP address. This typically causes a reverse DNS lookup to occur, though other sources of information such as */etc/hosts* may be used, depending on the configuration of your operating system.

If the lookup fails (that is, if Exim cannot find a hostname for the IP address), it takes a hard line by default and access is not permitted. If the list is an "accept" list (for example, `host_accept_relay`), Exim behaves as if the current host is not in the set defined by the list; if it is a "reject" list (for example, `host_reject`), it behaves as if it is.

One side effect of this is that subsequent items in the list are not examined, even if they do not require the hostname to be known.* For this reason, you should always put items involving IP addresses first if you can. Suppose you had the setting:

```
host_accept_relay = *.myfriend.example : 192.168.5.4
```

When a call from any host arrives, Exim has to find a hostname before it can test the first item in the list. If a name cannot be found, relaying is denied, even from the host 192.168.5.4. Putting the items in the opposite order allows relaying from that host, whether or not its name can be found. It is also more efficient, since the reverse lookup is not even attempted.

In some circumstances it may be necessary not to reject access if a reverse lookup fails. To allow this, the special item `+allow_unknown` may appear in a host list at top level; it is not recognized in an interpolated file. If any subsequent items require a hostname and the reverse lookup fails, Exim permits the access instead of rejecting it; that is, its behavior is the opposite to the default. For example:

```
host_reject = +allow_unknown:*.enemy.example
```

rejects connections from any host whose name matches `*.enemy.example`, but only if it can find a hostname from the incoming IP address. This is a dangerous thing to do if you really are dealing with a malicious enemy, because the block can easily be circumvented by unregistering the calling hosts. However, if you must accept mail from unregistered hosts but also need to block other registered hosts by name, this facility can be useful.

If `+warn_unknown` is used instead of `+allow_unknown`, the effect is the same, except that Exim writes an entry to its log when it accepts a host whose name it cannot look up.

* The order of items in a list matters because of the existence of negative items, so an unresolvable item cannot just be skipped.

As a result of aliasing, hosts may have more than one name. When processing any of the following items, all the host's names are checked:

- If an item in a host list starts with an asterisk, the remainder of the item must match the end of the hostname. For example, `*.b.c` matches all hosts whose names end in *.b.c*. The asterisk may appear only at the start of the item; this special simple form is provided because this is a very common requirement. Other kinds of wildcarding require the use of a regular expression (see the next bullet).
- If an item starts with a circumflex, it is taken to be a regular expression that is matched against the hostname. For example:

  ```
  ^[ab]\.c\.d$
  ```

  matches either of the two hosts *a.c.d* or *b.c.d*. As in the case of a domain list, the circumflex is interpreted as part of the regular expression.
- If an item is of the form *`search-type;filename-or-query`*, for example:

  ```
  host_reject = ! dbm;/host/accept/list
  ```

  the hostname is looked up using the search type and file name or query (as appropriate). If the lookup succeeds, the item matches. The retrieved data is not used.

Take care not to confuse lookups that use the hostname as the key with those that use the IP address. You must precede the search type with `net-` if you want the address to be used.

Use of RFC 1413 Identification in Host Lists

RFC 1413 (*Identification Protocol*) is much misunderstood. It defines a protocol (usually called *ident*) by which a server, on receiving a call from a client, can make an IP call back to the client and retrieve identification information relating to the original call. In the context of SMTP, the following sequence occurs:

1. Host C (client) connects to the SMTP port on host S (server) from an arbitrary port.
2. Host S connects to host C's ident port, passing host C's calling port.
3. Host C sends host S identification data relating to the SMTP call.
4. Host S records the data with the incoming message.

The misunderstanding some people have is to think that the information is supposed to be of use to the server. It is not; it is something the server can record and pass back to the client's administrator if there is a query.

Consider a large shared machine with thousands of registered users. If a user of that system makes a TCP/IP call to another host and abuses it in some way, the manager of the shared system, when investigating the resulting complaint, has a much easier task if the called host recorded RFC 1413 identification information obtained from the calling host. Many hosts simply send out login names in response to RFC 1413 calls, in which case the culprit is immediately identifiable. Some hosts include more information, such as time and source of login, and for privacy reasons, some hosts encrypt the information. RFC 1413 is an aid to finding the human responsible for a particular TCP/IP call from a multiuser system. It is of no use in the context of single-user clients, but that is not a problem because there should only be one human involved.

Exim makes RFC 1413 callbacks by default, but can be configured to do so only for particular hosts by setting `rfc1413_hosts`. This option defaults to:

```
rfc1413_hosts = *
```

that is, the callbacks are made for all hosts. A timeout is applied to these calls, controlled by `rfc1413_query_timeout`, which defaults to `30s` (30 seconds). If it is set to a zero length of time, no RFC 1413 calls are ever made. This is the recommended way of disabling these callbacks.

Any identification information that is received is logged with incoming messages, but in can also be used in other host lists. Any item in a host list other than an interpolated filename, `+allow_unknown` (or `+warn_unknown`) optionally can be preceded by:

`ident``@`
or
`!`*`ident`*`@`

where *`ident`* is an RFC 1413 identification string that must match the RFC 1413 identification sent by the remote host (unless it is preceded by an exclamation mark, in which case it must not match). The remainder of the item, following the `@`, may be either positive or negative. For example:

```
host_reject = !exim@my.gate.example : !root@public.host.example
```

rejects messages from *my.gate.example* unless the RFC identification is *exim*, and from *public.host.example* unless the RFC identification is *root*.

Address Lists

Address lists are used in a number of options to vary Exim's behavior for certain sender or recipient addresses. For example:

```
sender_reject_recipients = cleo@egypt.example : tony@egypt.example
```

specifies that any messages with those senders be refused by rejecting all their recipients.

Each item in an address list is matched against a mail address in the form *local_part@domain*. The following kinds of items may appear inline or as lines in an interpolated file:

- If an item starts with a circumflex, a regular expression match is done against the complete address, using the entire item as the regular expression. For example:

  ```
  sender_reject_recipients = ^(cleo|tony)@egypt\.example$
  ```

  As in the case of domain and host lists, the circumflex is interpreted as part of the regular expression.

- Otherwise, if there is no @ in the item, it is first matched against the domain part of the address being tested, the local part being ignored. This match is done exactly as for an entry in a domain list. For example, the item may begin with * or it may be a (partial) lookup (see the section "Domain Lists," earlier in this chapter). For example:

  ```
  sender_reject_recipients = *.rome.example
  ```

  matches all addresses whose domains end in *.rome.example*. For a fixed domain name or an asterisk wildcard, such as this example, if the domain does not match, the entire address does not match. However, when the item is a lookup and the domain does not match, a second lookup is done, this time with the entire subject address as the key, not just the domain (but with partial matching disabled). This means that an item such as:

  ```
  sender_reject_recipients = partial-dbm;/some/file
  ```

  can reference a single file whose keys are a mixture of complete domains, partial domains, and individual mail addresses. For example, this file:

  ```
  egypt.example
  *.rome.example
  caesar@rubicon.example
  ```

matches all addresses whose domains are *egypt.example*, *rome.example*, or any domain ending in *.rome.example*, as well as the specific address *caesar@rubicon.example*.*

- If the item starts with `@@`*`lookup-item`* (for example, `@@lsearch;/some/file`), the address being checked is split into a local part and a domain. The domain is looked up in the file. If it is not found, there is no match. If it is found, the data that is looked up from the file is treated as a colon-separated list of local part items, each of which is matched against the subject local part in turn. As in all colon-separated lists in Exim, a colon can be included in an item by doubling.

  The lookup may be partial, or may cause a search for a default keyed by `*`. The local part items that are looked up can be regular expressions, begin with `*`, or even be further lookups. They may also be negated independently. For example, with:

  ```
  sender_reject_recipients = @@dbm;/etc/reject-by-domain
  ```

  the data from which the DBM file is built could contain lines such as this:

  ```
  baddomain.example:  !postmaster : *
  ```

  If a sender's domain is *baddomain.example*, that line of local part items is retrieved and scanned. In this case, if the local part is *postmaster* it matches the negated item, so the whole address fails to match the list. Any other local part matches the asterisk. If a local part that actually begins with an exclamation mark is required, it has to be specified using a regular expression.

  If the last item in a list of local parts starts with a right angle bracket, the remainder of the item is taken as a new key to look up in order to obtain a continuation list of local parts. This is called *chaining*. The new key can be any sequence of characters. Thus, one might have entries such as:

  ```
  h1.example.com: spammer1 : spammer2 : >*
  h2.example.com: spammer3 : >*
  *:       ^\d{8}$
  ```

  to specify a match for eight-digit local parts for all domains, in addition to the specific local parts listed for each individual domain. Of course, using this feature costs another lookup each time a chain is followed, but the effort needed to maintain the data is reduced. It is possible to construct loops using this facility, and in order to catch them, the number of times Exim follows a chain is limited to 50.

* See the section "Partial Matching in Single-Key Lookups" in Chapter 16 for details of the partial matching facility for single-key lookups.

- If none of the earlier cases apply, the local part of the subject address is compared with the local part of the item, which may start with an asterisk. If the local parts match, the domains are compared in exactly the same way as entries in a domain list, except that a regular expression is not permitted. However, file lookups are allowed. For example:

  ```
  sender_reject_recipients = \
    *@*.spamming.site.example : \
    bozo@partial-lsearch;/list/of/dodgy/sites
  ```

  The domain may be given as a single `@` character, as in a domain list. A single `@` stands for the local hostname, leading to items of the form `user@@`. If a local part that actually begins with an exclamation mark is required, it has to be specified using a regular expression, as otherwise the exclamation mark is treated as a sign of negation.

Case of Letters in Address Lists

Domains in email addresses are always processed without regard to the case of any letters in their names. For local parts, the case may be significant on some systems.* However, RFC 2505 (*Antispam Recommendations for SMTP MTAs*) suggests that matching addresses to blocking lists should be done in a case-insensitive manner. Since most address lists in Exim are used for this kind of control, Exim attempts to do this by default.

The domain portion of an address is always made lowercase before matching to an address list. The local part is lowercase by default, and any string comparisons that take place are done case insensitively. This means that the data in the address list itself, in interpolated files, and in any file that is looked up using the `@@` mechanism, can be in either case. However, the keys in files that are looked up by a search type other than `lsearch` (which works case insensitively) must be in lowercase because these types of lookup are case sensitive.

To allow for case sensitive address list matching, if the string `+caseful` is included as an item in an address list, the original case of the local part is restored for any comparisons that follow, and string comparisons become case sensitive. This does not affect the domain.

* See the section "Handling Local Parts in a Case-Sensitive Manner," in Chapter 5, *Extending the Delivery Configuration*, to learn how Exim deals with this when processing local addresses.

19

Miscellany

This chapter collects a number of items that do not fit naturally into the other chapters, but are too small to warrant individual chapters of their own.

Security Issues

We use the word "security" to cover all aspects of the operation of Exim that are concerned with letting it perform privileged actions not permitted to ordinary user programs. It also covers aspects concerned with keeping the messages and other data it handles secure. There are three aspects to this:

- An MTA requires privilege to carry out the full range of expected functions, but it must take care to prevent its privilege from being abused. If possible, it should also relinquish privilege whenever it does not need it.
- An MTA must keep the files containing the messages it handles from being accessed by ordinary user programs. Under some countries' data protection legislation, messages and even mail logs are considered personal data, so it must be processed with appropriate care.
- An MTA must provide extra facilities for its administrators (for example, the ability to delete a message on the queue) that are safe from abuse by ordinary users.

Security is an important issue because breaches of security can lead to serious consequences. The full details of the security aspects of Exim are quite involved and allow for some variation in the way it is configured. However, there are some "standard" recommendations you should normally follow, unless you are sure you understand the consequences of doing otherwise. They are as follows:

- You should set up a special user (uid) and group (gid) for Exim's sole use. Many sites set up a user and a group called *exim*; others use *mail*. If you build Exim from source, define the uid and gid in your *Local/Makefile* as shown in this example:

  ```
  EXIM_GID=142
  EXIM_UID=142
  ```

  A description of how to build Exim is given in Chapter 22, *Building and Installing Exim*. If you are using a precompiled version of Exim that does not contain the settings you want to use, you can define the uid and gid in the runtime configuration file like this:

  ```
  exim_user = 142
  exim_group = 142
  ```

  This is slightly less safe because if those settings are lost, Exim may run under the wrong uid and gid.

- Add your system administrators to the Exim group. This allows them to read Exim's log files and carry out Exim administration functions without needing to know the *root* password; for more detail, see the discussion of privileged users later in this chapter.
- If you want your administrators to be able to run the *eximon* monitoring tool, they need to have read access to the message files on Exim's spool. The Exim group is set for these files, but the default mode is 0600, which gives access only to the Exim user (and *root*, of course). You need to change the mode to 0640; this can be done only if you build Exim from source. For details, see the section "Recommended Makefile Settings," in Chapter 22.
- Install Exim as *root*, using `make install`. This ensures that the binary is owned by *root* and has the setuid bit set.

The remainder of this section discusses security issues in more depth.

Use of Root Privilege

The way Exim uses the *root* privilege is quite complicated, and it can be configured to operate in more than one way. Before we go into the details, we'll give a brief review of the Unix features that are used.

How Unix uses uids to control privilege

From the start, Unix had the concept of a *real* uid and an *effective* uid for every process. These are both set to the same value when a user logs in. The effective uid is the one used for privilege checking (for example, file access), and it can be changed when a new program is run by setting the *setuid* flag in the owner permissions of the executable file. This causes the effective uid to be set to that of the file's owner; the real uid is unchanged. Modern versions of Unix also have a *saved* uid, which is set to the same value as the effective uid when the latter is changed at program startup.

The ways in which programs can manipulate these uids while they are running are not exactly the same in all versions of Unix, but a process whose effective uid is *root* is able to set all of them to any value, and any process can change its effective uid to its real or saved uid as and when it chooses. This means that there are two different ways in which a program with *root* privilege (one whose effective uid is *root*) can give up the privilege:

- If all three uids are set to something other than *root*, the abdication is permanent within the current program; privilege cannot be regained except by executing a new program that is owned by *root* and has the setuid flag set.
- If the real or saved uid is set to *root* and the effective uid is set to something else, the abdication is only temporary. Because any program may change its effective uid to its real or saved uid, privilege can be regained at any time.

Relinquishing the privilege temporarily is a less secure action because an error could cause it to be reinstated at the wrong time.*

Why does Exim need root privilege?

Exim does two things that requires it to be privileged:

- When started as a daemon, it sets up a socket connected to the SMTP port (port 25), which can be done only by a privileged program.
- It changes uid and gid to read forward files and run local delivery processes as the receiving user, or as the user specified in the configuration.

Because of these requirements, the Exim binary is normally setuid to *root*. In some special circumstances (for example, when the daemon is not in use and there are

* For those who know about Unix system functions: permanent abdication is implemented by calling `setuid()`, whereas temporary abdication is implemented by calling `seteuid()` (or, on a few systems, `setresuid()`).

no conventional local deliveries), it may be possible to run it setuid to some user other than *root* (usually the Exim user). This possibility is discussed later, but in the vast majority of Exim installations, the output of the *ls -l* command should look like this:

```
-rwsr-xr-x  1 root    smd       560300 Jun 14 08:53 exim
```

That is, the binary is owned by *root*, and the `s` flag is set for the owner. This means that whenever the program is run, the effective uid is changed to *root*. The group (`smd` in this case) is not normally relevant.

If no Exim user is specified in either the compile time or runtime configuration files, Exim runs as *root* all the time, except when performing local deliveries. Otherwise, it gives up *root* privilege when it no longer needs it and runs as the Exim user instead. In particular, it does this when receiving messages from any source, and also when making remote deliveries. This is why it is recommended that you define an Exim user.

Relinquishing root privilege

When Exim relinquishes *root* privilege, it may do so using either the permanent or temporary method described previously. In some cases, this can be controlled by a configuration option; this allows you to select a little less security to obtain a little more performance. To avoid cluttering up the text too much, we talk only about setting the uid in what follows. However, it should be understood that there is always a corresponding gid, and that whenever the uid is changed, the gid is changed also.

There are two circumstances in which an Exim process always gives up its privilege permanently:

- At the start of running a local delivery process. There are no exceptions. This applies whether or not an Exim user is defined.
- When a delivery process is about to do remote deliveries, provided an Exim user and group are defined. Local deliveries are done before remote deliveries, so *root* privilege is no longer needed.

There are also two instances in which Exim always gives up its privilege temporarily:

- When reading a user's *.forward* file. This is necessary when the file is not publicly readable and is on a remote NFS file system that is mounted without *root* privilege. If the file is a filter file, the effective uid remains unprivileged while being interpreted.

- If any director or router has the `require_files` option set to check the existence of a file as a specific user, the effective uid is changed to that user for the duration of the check.

The other circumstances in which privilege is relinquished require an Exim user and group to be defined. If you have not set these up, Exim runs as *root*, except as just described, because it has no other uid it can use.

The security option

The `security` configuration option controls whether Exim relinquishes its privilege permanently or temporarily in these remaining circumstances. You should not normally need to change the default setting (which is the most secure) unless you are running an unusual configuration or are prepared to sacrifice a little security in exchange for less resource usage.

- With the setting:

  ```
  security = seteuid
  ```

  Exim gives up *root* temporarily when it does not need it (for example, while running the routers and directors) and regains the privilege when necessary. This enables it to run with a non-*root* effective uid most of the time, at very little cost, but offers less security.

- With the setting:

  ```
  security = setuid
  ```

  Exim gives up its privilege permanently when it is receiving a locally generated message and after it has set up a listening socket when running as a daemon. This means that, to deliver any message that it has received, it has to reinvoke a fresh copy of itself to regain privilege. During delivery, it retains privilege except when actually transporting the message. In particular, it runs the directors and routers as *root*.

- With the setting:

  ```
  security = setuid+seteuid
  ```

  which is the default (provided an Exim user and group are defined), Exim operates as for `setuid`, but it also gives up privilege temporarily when it needs to regain it subsequently without losing a lot of state information (for example, while running the directors and routers).

The most secure setting necessarily involves the use of more resources because the Exim binary has to be reinvoked more frequently, but on a busy system it is likely to remain in the file system's cache, so the cost is probably not that large.

Running Local Deliveries as root

It is generally considered to be a bad idea to run any local deliveries as *root* on the grounds of avoiding excessive privilege where it is not needed. Most installations set up *root* as an alias for the system administrator, which bypasses this problem, but just in case this is not done, Exim's default configuration contains a "trigger guard" in the form of the setting:

```
never_users = root
```

Whenever Exim is about to run a local delivery process, it checks to see if the required uid is one of those listed in `never_users`. If it is, the delivery is run as *nobody* instead. The uid and gid for *nobody* can be specified by `nobody_user` and `nobody_group`; the default is to look up the login name *nobody*.

Running an Unprivileged Exim

In some restricted environments, it is possible to run Exim without any privilege at all, or by retaining privilege only when starting a daemon process. This gives added security, but restricts the actions Exim is able to take. A host that does no local deliveries is a good candidate for this kind of configuration. There are two possibilities if you want to run Exim in this way:

- Keep its setuid to *root*, as in other configurations, but set:

  ```
  security = unprivileged
  ```

  In all cases, except when starting the daemon, this setting causes Exim to give up privilege permanently as soon as it starts, and thereafter it runs under the Exim uid and gid. In the case of the daemon, *root* privilege is retained only until Exim has bound its listening socket to the SMTP port. The daemon can respond correctly to a SIGHUP signal requesting that it reload its configuration because the reinvocation that this causes regains *root* privilege.

- Make the Exim binary setuid and setgid to the Exim user and group:

  ```
  -rwsr-sr-x  1 exim   exim     560300 Jun 14 08:53 exim
  ```

  This means that it always runs under the Exim uid and gid and cannot start up as a daemon unless it is called by a process that is running as *root*. A daemon cannot restart itself as a result of SIGHUP because it is no longer a *root* process at that point. You should still set:

  ```
  security = unprivileged
  ```

  in this case, because this setting stops Exim from trying to reinvoke itself to do a delivery after a message has been received. Such a reinvocation is a waste of time because it would have no effect. Instead, Exim just carries on in the same incarnation of the program.

If the second approach is chosen, unless Exim is invoked from a *root* process, it ends up running with the real uid and gid set to those of the invoking process, and the effective uid and gid set to Exim's values. Ideally, any association with the values of the invoking process should be dropped; that is, the real (and saved) uid and gid should be reset to the effective values. Some operating systems have a function that permits this action for a non-*root* effective uid, but quite a number of them do not. Because of this lack of standardization, Exim does not address this problem. For this reason, the first approach is perhaps the better one to take if you are concerned about this issue.

The `unprivileged` setting of the `security` option is more efficient than `setuid` or `setuid+seteuid` because Exim no longer needs to reinvoke itself when starting a delivery process after receiving a message. However, to achieve this extra efficiency, you have to submit to some restrictions, which are all concerned with handling local addresses and local deliveries. There are no special restrictions on message reception or remote (SMTP) delivery.

The restrictions are as follows:

- All local deliveries are run under the Exim uid and gid. You should explicitly use the `user` and `group` options to override directors or transports that normally deliver as the recipient. This use allows configurations that work in this mode to function the same way with other `security` settings. Any implicit or explicit specification of a different user causes an error.
- Use of *.forward* files is severely restricted, such that it is usually not worthwhile to include a **forwardfile** director in the configuration. Users who wished to use *.forward* would have to make their home directories and the files themselves accessible to the Exim user. Even if this is done, pipe and file items in *.forward* files, and their equivalents in Exim filters, cannot be permitted in practice. Although such deliveries could be allowed to run as the Exim user, that would be insecure and probably not very useful.
- Unless user mailboxes are all owned by the Exim user, which is possible in some POP3-only or IMAP-only environments:
  - They must be owned by the Exim group and be writable by that group. This implies that you must set `mode` in the **appendfile** configuration, as well as the mode of the mailbox files themselves.
  - You must set `no_check_owner` in the **appendfile** configuration, since most or all of the files will not be owned by the Exim user.
  - You must set `file_must_exist` in the **appendfile** configuration, as Exim cannot set the owner correctly on a new mailbox when unprivileged. This also implies that new mailboxes must be created manually.

Privileged Users

A privileged user is one who is permitted to ask Exim to do things that normal users may not. There are two different kinds of action involved, so there are two different classes of privileged users, called *trusted* users and *admin* users. In the descriptions of the command-line options in Chapter 20, *Command-Line Interface to Exim*, a restriction to trusted or admin users is noted for those options to which it applies.

Trusted Users

Trusted users are allowed to override certain information when submitting messages via the command line (that is, other than over TCP/IP). The Exim user and *root* are automatically trusted and additional trusted users can be defined by the `trusted_users` option, for example:

```
trusted_users = uucp : majordom
```

In addition, if the current group or any of the supplementary groups of the process that calls Exim is the Exim group, or any group listed in the `trusted_groups` option, the caller is trusted.

Setting the sender of a locally submitted message

When an ordinary (nontrusted) user submits a message locally, a sender address is constructed from the login name of the real user of the calling process and the default qualifying domain. This address is set as the sender in the message's envelope. It is also placed in an added *Sender:* header line if the *From:* header does not contain it, though this can be disabled by setting `no_local_from_check`.*

A trusted user may override the sender address by using the *-f* option. For example:

```
exim -f 'alice@carroll.example' ...
```

forces the sender address to be *alice@carroll.example*. If you are running mailing list software that is external to Exim, you should arrange for it to run as a trusted user so it can specify sender addresses when passing messages to Exim for delivery to subscribers.

* There is more discussion of the *Sender:* header line in the section "The Sender: Header Line," in Chapter 13, *Message Reception and Policy Controls*.

The origin of the concept of trusted users lies in multiuser systems, where the administration wants to ensure that an authenticated sender address is present in every message that is sent out.[*] In this kind of environment, trusted users are those able to "forge" sender addresses when submitting messages using the command-line interface.

On small workstations where everything that is done can be accounted to a few people, the distinction between trusted and nontrusted users is less useful, expecially in the case of sender addresses. If you are running such a system, you may want to remove the restriction on the use of *-f*. From Release 3.20, you can do this by setting the `untrusted_set_sender` option. This does not, however, make all users trusted; it applies only to the use of *-f*.

Setting other information in a locally submitted message

In addition to a sender address, a trusted user may supply additional information, such as an IP address, as if the message had been received from a remote host, using command-line options described in the section "Additional Message Data," in Chapter 20. These facilities are provided to make it possible for administrators to inject messages with "remote" characteristics using the command line. This can be useful when passing on messages that have arrived via some other transport system, such as UUCP, or when reinjecting messages that initially have been delivered to a virus scanner (for example).

Admin Users

Admin users are permitted to use options that affect the running of Exim, for example, to start daemon or queue runner processes and to remove messages from the queue. The Exim user and *root* are automatically admin users and additional admin users can be set up by adding them to the Exim group.

If you want to, you can "open up" two actions normally permitted only to admin users so that any user can request them:

- If `no_prod_requires_admin` is set, any user may start an Exim queue run by means of the *-q* option, and may also request the delivery of an individual message by means of the *-M* option.
- If `no_queue_list_requires_admin` is set, any user may list the messages on the queue by means of the *-bp* option. Otherwise, nonadmin users see only the messages that they themselves have submitted.

[*] For messages sent directly over TCP/IP from user processes, the `ident` protocol can help provide similar accountability. See the section "Use of RFC 1413 Identification in Host Lists," in Chapter 18, *Domain, Host, and Address Lists*.

If you want to make all members of an existing group into admin users, you can do so by specifying the group in the `admin_groups` option. The current group does not have to be one of these groups in order for an admin user to be recognized. For example, setting:

```
admin_groups = sysadmin
```

makes every user in the *sysadmin* group an Exim admin user. However, there is an advantage in doing it the other way; that is, in adding all your administrators to the *exim* group explicitly. If you do this, and if you arrange for Exim's spool and log files to have mode 0640, it gives the administrators read access to these files, which is necessary if they want to run the *eximon* monitor program or examine log files directly.

RFC Conformance

The main RFCs that define basic Internet mail services are now very old. RFCs 821 and 822 were published in 1982; some clarifications were published in RFC 1123 in 1989. Subsequent RFCs have mostly been concerned with adding functionality such as MIME and extending the SMTP protocol.

The Internet has changed dramatically since 1982 and MTAs have had to change with it, in some cases adopting new conventions that are not in the RFCs, and in others choosing to ignore the RFCs' recommendations or relax their restrictions. Some, but not all, of these changes have been incorporated into revised versions of RFCs 821 and 822. These have been in preparation for some years, and at the time of writing are close to becoming Internet standards.

It is important to remember that the RFCs are not legally binding contracts; their intent is to facilitate widespread interworking over the network. If software conforms to the relevant RFCs, the chances that it can interwork successfully are high. However, you may find that not following an RFC in some particular instance extends interworking between your host and those with which it communicates. If such a change is widely adopted, it may eventually be sanctioned as a standard, though there are some cases where widely used practice is frowned on by the purists. Any particular piece of software must steer a middle course between strict adherence to the RFCs on the one hand, and total disregard on the other.

There are a number of ways in which Exim does not conform strictly to the RFCs. Some of them are very minor, others you can control by setting options, and a few are fundamental to the way the program works and cannot be disabled. These distinctions reflect the prejudices of the author.

8-Bit Characters

Although TCP/IP has always been an 8-bit transport medium, the mail RFCs, even in the forthcoming revision, still insist that mail is basically a 7-bit service. Characters with the most significant bit set (that is, with a value greater than 127) are forbidden.

The transfer of 8-bit material can be negotiated in some circumstances, but otherwise an MTA is supposed to encode 8-bit characters in some way before transmitting them. Note that this does not apply to binary attachments (which are already encoded into 7-bit characters by the MUA that creates the message), but rather to "raw" 8-bit characters received by the MTA. The most common reason why these are encountered is the use of accented and other special letters in European languages and names.

Requiring an MTA not to pass on 8-bit characters without special action raises technical problems and issues of design principle. If an MTA has received a message containing 8-bit characters and the remote MTA to which it wants to send the message has not indicated support for 8-bit transfers (which is an SMTP extension), the sending MTA must choose between three possibilities:

- Bounce the message.
- Translate the message into a 7-bit format, making an arbitrary choice of encoding mechanism.
- Just send the 8-bit characters anyway.

Strict adherence to the RFCs permits only the first two of these. However, the first is not very helpful, and the second may well turn the message into a form that is not displayed correctly to the final recipient.* Breaking the rules, however, and just sending the 8-bit characters as they are has a high probability of achieving the result that is intended: namely, the transfer of these characters from sender to recipient.

To make any decisions about 8-bit characters, an MTA has to at least check a message's body for their presence. Some people (including this author) feel strongly that the job of an MTA is to move messages about, not to spend resources inspecting or modifying their content. For this reason, Exim is "8-bit clean." It makes no modification to message bodies and it pays no attention to 8-bit characters contained therein; they are transported unmodified.

There is, however, an option that is concerned with 8-bit characters. When Exim acts as a server, it happily accepts 8-bit characters in messages in accordance with the philosophy just described, but not all clients are prepared to send such

* Converting messages into "quoted-printable" format is notorious for this.

characters in the way Exim does. The RFCs specify an SMTP extension called 8BITMIME for the transmission of 8-bit data. If the following:

```
accept_8bitmime
```

is set in Exim's runtime configuration file, it advertises the 8BITMIME extension in its response to the EHLO command. This causes certain clients to send 8-bit data unmodified instead of encoding it; they use the BODY= parameter on MAIL commands to indicate this. Exim recognizes this parameter, but it does not affect its actions in any way. Thus, setting accept_8bitmime is just a way of persuading clients not to encode 8-bit data. Exim processes such data in the same way, whether or not the BODY parameter is used.

Address Syntax

Syntactically invalid addresses, both in envelopes and header lines, are a depressingly common occurrence. Exim performs syntax checks on all RFC 821 addresses received in SMTP commands, but it does not check header lines unless headers_check_syntax is set, or it is extracting envelope addresses from header lines as a result of the *-t* option.

Built-in address syntax extensions

A number of extensions to the syntax specified in the RFCs are always permitted:

1. Exim accepts a header line such as:

   ```
   To: A.N.Other <ano@somewhere.example>
   ```

   which is strictly invalid because dot is a special character in RFC 822; the line really should be:

   ```
   To: "A.N.Other" <ano@somewhere.example>
   ```

   but many mail programs allow the unquoted form.

2. Strictly, a local part may not end with a dot, nor contain adjacent dots, for example:

   ```
   P.H.@somewhere.example
   A..Z@somewhere.example
   ```

   Again, these forms are accepted because of widespread use.

3. "Quoted pairs" in unquoted local parts, for example:

   ```
   abc\@xyz@somwhere.example
   ```

   are permitted by RFC 821 but not by RFC 822; for simplicity, Exim always accepts them.

4. When reading SMTP `MAIL` and `RCPT` commands, Exim does not insist that addresses be enclosed in angle brackets.

Configurable address syntax extensions

Two other address extensions can be enabled by setting options:

1. Misconfigured mailers occasionally send out addresses within additional pairs of angle brackets, for example:

   ```
   MAIL FROM:<<xyz@bad.example>>
   ```

   This normally causes a syntax error, but if `strip_excess_angle_brackets` is set, the excess brackets are removed by Exim.

2. The DNS use of a trailing dot to indicate a fully qualified domain sometimes causes confusion and leads to the use of mail addresses that end with a dot, for example:

   ```
   dotty@dot.example.
   ```

   Again, this normally causes a syntax error, but if `strip_trailing_dot` is set, the trailing dot is quietly removed.

Domain literal addresses

The RFCs permit the use of *domain literal* addresses, which are of the form:

```
shirley@[10.8.3.4]
```

That is, instead of a domain name, an IP address enclosed in brackets is used. This causes the message to be sent to the host whose IP address it is. Even in 1982, when RFC 822 was written, the use of domain literals was recognized as undesirable. The RFC says:

> The use of domain-literals is strongly discouraged. It is permitted only as a means of bypassing temporary system limitations, such as name tables which are not complete.

In the modern Internet, addressing messages to specific hosts by their IP addresses is seen by many as utterly undesirable. For this reason, Exim's default configuration file contains:

```
forbid_domain_literals
```

which causes Exim not to recognize the syntax of domain literal addresses. If you want to permit the use of these addresses, you have to remove this setting from the configuration and ensure that there is an **ipliteral** router to do the routing.

If you want to recognize incoming messages containing domain literal addresses for your own host, you must either include them in `local_domains` in domain literal format, for example:

```
local_domains = myhost.example : [192.168.10.8]
```

Alternatively, you can set:

```
local_domains_include_host_literals
```

which causes all the IP addresses for your host to be added to `local_domains` automatically.

Source routed addresses

Finally, while on the subject of address syntax, Exim recognizes so-called "source routed" addresses of the form:

```
@relay1,@relay2,@relay3:user@domain
```

However, the use of such addresses has been discouraged since RFC 1123, and an MTA is entitled to ignore all the routing information and treat such an address as:

```
user@domain
```

This is what Exim does.

Canonicizing Addresses

When a mail domain is the name of a CNAME record in the DNS, the original RFCs suggest that an MTA should automatically change it to the "canonical name" as a message is processed, and there are MTAs that do make this change. However, the forthcoming revised RFCs do not contain this suggestion, and Exim does not perform this rewriting.

Coping with Broken MX Records

The righthand side of an MX record is defined as a hostname. Some DNS administrators fail to appreciate this and set up MX records with IP addresses on the righthand side, like this:

```
clueless.example.   MX  1   192.168.43.26
```

Unfortunately, there are broken MTAs in use that do not object to these records, leading people to think that they will always work. Exim is not one of them; it treats the righthand side as a domain name, tries to look up its address records, and naturally fails.*

If you are in a situation in which you just have to deliver mail to such domains (perhaps you want to send a message to a postmaster pointing out the error), you can get Exim to misbehave by setting:

```
allow_mx_to_ip
```

in its configuration. This setting is not recommended for general use.

Line Terminators in SMTP

The specification of SMTP states that lines are terminated by the two-character sequence consisting of carriage return (CR) followed by linefeed (LF), and this is what Exim uses when it sends out SMTP. For incoming SMTP, however, there are clients known to break the rules and just use linefeed alone to terminate lines. For this reason, Exim accepts such input.

Syntax of HELO and EHLO

One of the more common errors in client SMTP implementations is sending syntactically invalid `HELO` or `EHLO` commands. Exim accepts underscores in the argument by default because this is an extremely widespread practice, but rejects other syntax errors. There are some options that can be used to change this behavior; they are described in the section "Verifying EHLO or HELO," in Chapter 13.

Timestamps

Exim uses a timestamp for every line it writes to any of its log files and for every *Received:* header it creates. By default, these timestamps are in the local wallclock time zone, but there are two ways you can change this:

- If you set `timestamps_utc` in Exim's runtime configuration, all timestamps are in the Universal Coordinated Time (UTC, also known colloquially as GMT).
- Otherwise, the setting of the `timezone` option controls which time zone is used. For example, if you set:

  ```
  timezone = EST
  ```

  timestamps are in Eastern Standard Time.

Unfortunately, there is apparently no standard way a Unix program can specify the use of local wallclock time without knowing what the local time zone is. It can be

* It does, however, notice what is going on and adds a suitable comment to the failure message.

done in some operating systems, but not in others. For this reason, the default setting for `timezone` is taken from the setting of the `TZ` environment variable at the time Exim is built, in the hope that this does the right thing in most cases.

If `TZ` is unset when Exim is built or if `timezone` is set to the empty string, Exim removes any existing `TZ` variable from the environment when it is called. On Linux, Solaris, and BSD-derived operating systems, this causes wallclock time to be used.

Checking Spool Space

In the section "Incoming SMTP Messages over TCP/IP," in Chapter 15, *Authentication, Encryption, and Other SMTP Processing*, the checking of available spool space was mentioned in connection with the `SIZE` option of the `MAIL` command. This is just one particular case in which this happens. There are four options that request checks on disk resources before accepting a new message.

If either `check_spool_space` or `check_spool_inodes` contains a value greater than zero, for example:

```
check_spool_space = 50M
check_spool_inodes = 100
```

the free space in the disk partition that contains the spool directory is checked to ensure that there is at least as much free space and as many free inodes as specified. The check happens at the start of accepting a message from any source. The check is not an absolute guarantee because there is no interlocking between processes handling messages that arrive simultaneously.

If you have configured Exim to write its log files in a different partition to the spool files, you can set `check_log_space` and `check_log_inodes` in the same way to check that partition.

If there is less space or fewer inodes than requested, Exim refuses to accept incoming mail. In the case of SMTP input, this is done by giving a 452 temporary error response to the `MAIL` command. If there is a `SIZE` parameter on the *MAIL* command, its value is added to the `check_spool_space` value and the check is performed even if `check_spool_space` is zero, unless `no_smtp_check_spool_space` is set.

For non-SMTP input and for batched SMTP input, the test is done at startup; on failure a message is written to the standard error stream and Exim exits with a non-zero code, as it obviously cannot send an error message of any kind.

Control of DNS Lookups

In a conventional configuration, Exim makes extensive use of the DNS when handling mail. Most of the time this "just happens," and you do not need to be concerned with the details of DNS lookups. However, DNS problems are not entirely unknown; they can sometimes be alleviated by changing the way Exim does its DNS lookups.

Whereas the DNS itself can store domain names that contain almost any character, domains used in email are restricted to letters, digits, hyphens, and dots. Some DNS resolvers have been observed to give temporary errors if asked to look up a domain name (for an MX record, say) that contains other characters. To avoid this problem, Exim checks domain names before passing them to the resolver by matching them against a regular expression specified by `dns_check_names_pattern`. The default setting is:

```
dns_check_names_pattern = \
  (?i)^(?>(?(1)\.|())[^\W_](?>[a-z0-9-]*[^\W_])?)+$
```

which permits only letters, digits, and hyphens in domain name components, and requires them neither to start nor end with a hyphen. If a name contains illegal characters, Exim behaves as if the DNS lookup had returned "not found." This checking behavior can be suppressed by setting `no_dns_check_names`.

Badly set-up name servers have been seen to give temporary errors for domain lookups for long periods of time. This causes messages to remain on the queue and be retried until they time out. Sometimes you may know that a particular domain does not exist. If you list such domains in `dns_again_means_nonexist`, a temporary DNS lookup error is treated as a nonexistent domain, causing messages to bounce immediately. This option should be used with care because there are many legitimate cases of temporary DNS errors.

Finally, the options `dns_retrans` and `dns_retry` can be used to set the retransmission and retry parameters for DNS lookups. Values of zero (the defaults) leave the system default settings unchanged. The value of `dns_retrans` is the time in seconds between retries and `dns_retry` is the number of retries. Exactly how these settings affect the total time a DNS lookup may take is not clear.

Bounce Message Handling

This section covers several options that alter the way Exim handles or generates bounce messages (that is, delivery failure reports). This also includes warning messages, which are sent after a message has been on the queue for a specific time. Warning messages have the same format as bounce messages.

Replying to Bounce Messages

When Exim generates a bounce message, it inserts a *From:* header line specifing the sender as *Mailer-Daemon* at the default qualifying domain. For example:

```
From: Mail Delivery System <Mailer-Daemon@myhost.example>
```

Experience shows that many people reply, either accidentally, or out of ignorance, to such messages.* You should normally arrange for *mailer-daemon* to be an alias for *postmaster* if you want to see these messages. Another thing you can do is to set `errors_reply_to`, which provides the text for a *Reply-to:* header line. For example:

```
errors_reply_to = postmaster@myhost.example
```

Taking Copies of Bounce Messages

Sometimes there is a requirement to monitor the bounce messages that Exim is creating. The `errors_copy` option can be used to cause copies of bounce messages sent to particular addresses to be copied to other addresses. The value is a colon-separated list of items; each item consists of a pattern and a list of addresses, separated by whitespace. If the pattern matches the recipient of the bounce, the message is copied to the addresses on the list. The items are scanned in order, and once a match is found, no further items are examined. For example:

```
errors_copy = spqr@mydomain   postmaster@mydomain :\
              rqps@mydomain   postmaster@mydomain,\
                              hostmaster@mydomain
```

takes copies of any bounces sent to *spqr@mydomain* or *rqps@mydomain*. The bounces are copied to *postmaster@mydomain* in both cases; those for *rqps@mydomain* are also copied to *hostmaster@mydomain*. To send copies of all bounce messages to the postmaster you could use:

```
errors_copy = *@*  postmaster
```

Each pattern can be a single regular expression, indicated by starting it with a circumflex; alternatively, either portion (local part or domain) can start with an asterisk, or the domain can be in any format that is acceptable as an item in a domain list, including a file lookup. A regular expression is matched against the entire (fully qualified) recipient; nonregular expressions must contain both a local part and domain.

* In addition, some software, in contravention of the RFCs, generates automatic replies to bounce messages by extracting an address from the header lines.

The address list is a string that is expanded, and must end up as a comma-separated list of addresses. The expansion variables $local_part and $domain are set from the original recipient of the error message, and if there is any wildcard matching, the expansion variables $0, $1, etc. are set in the normal way.

Messages to Postmaster

Exim sends a message to the local postmaster in certain circumstances, and the address it uses is specified by the `errors_address` option, defaulting to:

```
errors_address = postmaster
```

In current versions of Exim, the only common occasion on which this is used is when a nonbounce message is frozen, if `freeze_tell_mailmaster` is set.* No message is sent when a bounce message is frozen because of the possibility that this might cause a loop, so `freeze_tell_mailmaster` is not useful for alerting the postmaster of these incidents.

Delay Warning Messages

When a message is delayed (that is, if it remains on Exim's queue for a long time), Exim sends a warning message to the sender at intervals specified by the `delay_warning` option, provided certain conditions (described later) are met. The default value for `delay_warning` is 24 hours; if it is set to a zero time interval, no warnings are sent. The data is a colon-separated list of times after which to send warning messages. Up to ten times may be given. If a message has been on the queue for longer than the last time, the last interval between the times is used to compute subsequent warning times. For example, with:

```
delay_warning = 1h
```

warnings are sent every hour, whereas with:

```
delay_warning = 4h:8h:24h
```

the first message is sent after 4 hours, the second after 8 hours, and subsequent ones every 16 hours thereafter. To stop warnings after a given time, set a huge subsequent time, for example:

```
delay_warning = 4h:24h:99w
```

Nowadays, when most messages are delivered very rapidly, users appreciate warnings of delay, but on the whole they do not usually like them to be repeated too

* It can also occur if a bounce message fails for any reason other than timeout, but that happens only if such a message is failed by the *-Mg* command-line option.

often. n the other hand, sending such warnings to the managers of mailing lists is usually counterproductive. To ensure that warnings are sent only in appropriate circumstances, Exim has the `delay_warning_condition` option.

This string is expanded at the time a warning message might be sent. If all the deferred addresses have the same domain, it is set in $domain during the expansion; otherwise $domain is empty. If the result of the expansion is a forced failure, an empty string, or a string matching any of `0`, `no`, or `false` (the comparison being done caselessly), the warning message is not sent. The default setting for the option is:

```
delay_warning_condition = \
  ${if match{$h_precedence:}{(?i)bulk|list|junk}{no}{yes}}
```

which suppresses the sending of warnings for messages that have `bulk`, `list`, or `junk` in a *Precedence:* header line. This covers most mailing lists.

Customizing Bounce Messages

Default text for the message that Exim sends when an address is bounced is built into the code of Exim, but you can change it, either by adding a single string or by replacing each of the paragraphs by text supplied in a file.

If `errmsg_text` is set, its contents, which are not expanded, are included in the default message immediately after "This message was created automatically by mail delivery software." For example:

```
errmsg_text = If you don't understand it, please ask your postmaster \
              for help.
```

Alternatively, you can set `errmsg_file` to the name of a template file for constructing error messages. The file consists of a series of text items, separated by lines consisting of exactly four asterisks. If the file cannot be opened, default text is used and a message is written to the main and panic logs. If any text item in the file is empty, default text is used for that item.

Each item of text that is read from the file is expanded, and there are two expansion variables that can be of use here: $errmsg_recipient is set to the recipient of an error message while it is being created, and $return_size_limit contains the value of the `return_size_limit` option, rounded to a whole number. The items must appear in the file in the following order:

- The first item is included in the header lines of the bounce message and should include at least a *Subject:* header. Exim does not check the syntax of these lines.

- The second item forms the start of the error message. After it, Exim lists the failing addresses with their error messages.
- The third item is used to introduce any text from **pipe** transports that is to be returned to the sender. It is omitted if there is no such text.
- The fourth item is used to introduce the copy of the message that is returned as part of the error report.
- The fifth item is added after the fourth one if the returned message is truncated because it is bigger than `return_size_limit`.
- The sixth item is added after the copy of the original message.

The default state (`errmsg_file` unset) is equivalent to the following file, in which the sixth item is empty. The *Subject:* line has been split to fit it on the page:

```
Subject: Mail delivery failed
  ${if eq{$sender_address}{$errmsg_recipient}{: returning message to sender}}
****
This message was created automatically by mail delivery software.

A message ${if eq{$sender_address}{$errmsg_recipient}{that you sent }{sent by

  <$sender_address>

}}could not be delivered to all of its recipients.
This is a permanent error. The following address(es) failed:
****
The following text was generated during the delivery attempt(s):
****
------ This is a copy of the message, including all the headers. ------
****
------ The body of the message is $message_size characters long; only the first
------ $return_size_limit or so are included here.
****
```

Customizing Warning Messages

The text of delay warning messages (those sent as a result of the `delay_warning` option) can be customized in a similar manner to bounce messages. You can set `warnmsg_file` to the name of a template file, which in this case contains only three text sections:

- The first item is included in the header lines and should include at least a *Subject:* header. Exim does not check the syntax of these lines.

- The second item forms the start of the warning message. After it, Exim lists the delayed addresses.
- The third item ends the message.

During the expansion of this file, $warnmsg_delay is set to the delay time in one of the forms *n* `minutes` or *n* `hours`, and $warnmsg_recipients contains a list of recipients for the warning message. There may be more than one recipient if multiple addresses have different `errors_to` settings on the routers or directors that handled them. The default state is equivalent to the file:

```
Subject: Warning: message $message_id delayed $warnmsg_delay
****
This message was created automatically by mail delivery software.

A message ${if eq{$sender_address}{$warnmsg_recipients}{that you sent }{sent by

  <$sender_address>

}}has not been delivered to all of its recipients after
more than $warnmsg_delay on the queue on $primary_hostname.

The message identifier is:     $message_id
The subject of the message is: $h_subject
The date of the message is:    $h_date

The following address(es) have not yet been delivered:
****
No action is required on your part. Delivery attempts will continue for
some time, and this warning may be repeated at intervals if the message
remains undelivered. Eventually the mail delivery software will give up,
and when that happens, the message will be returned to you.
```

Miscellaneous Controls

This section contains brief descriptions of some minor options that do not merit a section to themselves, but that may be of general interest. The Exim reference manual describes additional, minority-interest options.

`local_domains_include_host` (Boolean, default = false)
: If this option is set, the value of `primary_hostname` is added to the value of `local_domains`, unless it is already present. This makes it possible to use the same configuration file on a number of different hosts. The same effect can be obtained by including the conventional item `@` (which matches the primary host name) in `local_domains`.

`max_username_length` (integer, default = 0)
: Some operating systems are broken so they truncate the argument to `getpw-nam()` (the function that reads information about a login name) to eight characters, instead of returning "no such user" for longer names. If this option is

set to greater than zero, any attempt to call `getpwnam()` with an argument that is longer than its value behaves as if `getpwnam()` failed.

`received_headers_max` (integer, default = 30)
When a message is to be delivered, the number of *Received:* headers is counted. If it is greater than this parameter, a mail loop is assumed to have occurred, the delivery is abandoned, and an error message is generated. This applies to both local and remote deliveries.

`smtp_banner` (string, default = built-in)
This string, which is expanded every time it is used, is output as the initial positive response to an SMTP connection. The default setting is:

```
smtp_banner = $primary_hostname ESMTP Exim \
  $version_number #$compile_number $tod_full
```

Failure to expand the string causes Exim to write to its panic log and exit immediately. If you want to create a multiline response to the initial SMTP connection, put the string in quotes and use `\n` in the string at appropriate points, but not at the end. Note that the 220 code is not included in this string. Exim adds it automatically (several times in the case of a multiline response).

`smtp_receive_timeout` (time, default = 5m)
This sets a timeout value for SMTP reception. If a line of input (either an SMTP command or a data line) is not received within this time, the SMTP connection is dropped and the message is abandoned. For non-SMTP input, reception timeout is controlled by `accept_timeout`.

20

Command-Line Interface to Exim

Whenever Exim is called, it is passed options and arguments specifying what the caller wants it to do. Because you can call Exim from a shell in this way, this is called the *command-line interface*. In practice, most calls of Exim come directly from other programs such as MUAs, and do not involve an actual "command line." However, the options and arguments are the same.

Many command-line options are compatible with Sendmail, so Exim can be a drop-in replacement, but there are additional options specific to Exim. Some options can be used only when Exim is called by a privileged user, and these are noted in what follows.

The command-line options are many, but they can be divided into a number of functional groups as follows:

Input mode control
: Options to start processes for receiving incoming messages

Additional message data
: Options to supply information to be incorporated into an incoming message that is submitted locally

Immediate delivery control
: Options to control whether a locally submitted message is delivered immediately on arrival, possibly depending on the type of recipient addresses

Error routing
: Options to control how errors in a locally submitted message are reported

Queue runner processes
: Options for starting queue runners and selecting which messages they process

Configuration overrides
: Options for overriding the normal configuration file

Watching Exim
: Options for inspecting messages on the queue

Message control
: Options for forcing deliveries and doing other things to messages

Testing
: Options for testing address handling, filter files, string expansion, and retry rules

Debugging
: Options for debugging Exim and its configuration

Internal
: Options that are only useful when one instance of Exim calls another

Miscellaneous
: A few oddities

Compatibility with Sendmail
: Options that are recognized because they are used by Sendmail, but which do nothing useful in Exim

In this chapter, we'll discuss the options by functional group. You can find a complete list of options in alphabetical order in the reference manual.

Input Mode Control

Four mutually exclusive options control the way messages are received.

Starting a Daemon Process

If *-bd* is specified, a daemon process is started in order to receive messages from remote hosts over TCP/IP connections, using the SMTP procotol. It normally listens on the SMTP port (25), but the *-oX* option can be used to specify a different port. For example:

```
exim -bd -oX 1225
```

starts up a daemon that listens on port 1225. This can be useful for some nonstandard applications, and also for testing Exim as a daemon without disturbing the running mail service. Only admin users are permitted to start daemon processes.

Interactive SMTP Reception

If *-bs* is specified, a reception process starts that reads SMTP commands on its standard input and writes the responses to its standard output. This option can be used by any local process; however, if any messages are submitted during the SMTP session, the senders supplied in the `MAIL` commands are ignored unless the caller is trusted. The *-bs* option is also used to start up reception processes from *inetd* to receive mail from remote hosts as an alternative to using a daemon. The necessary entry in */etc/inetd.conf* should be along these lines:

```
smtp stream tcp nowait /usr/exim/bin/exim in.exim -bs
```

Exim can tell the difference between the two uses of *-bs* by testing its standard input to see whether it is a socket or not. If there is an associated IP address, the input must be a socket and the process must have been started by *inetd*. In this case, sender addresses from `MAIL` commands are honored.

Batch SMTP Reception

If *-bS* is specified, a reception process is started that reads SMTP commands on its standard input, but does not write any responses. This is so-called "batch SMTP," which is really just another way of injecting messages in a noninteractive format. This is commonly used for messages received by other transport mechanisms, such as UUCP, or for messages that have been stored in files temporarily. Once again, the senders supplied in the `MAIL` commands are ignored unless the caller is trusted. More details about the handling of all forms of SMTP are given in Chapter 15, *Authentication, Encryption, and Other SMTP Processing.*

Non-SMTP reception

If *-bm* is specified, a reception process is started that reads the body of the message from the standard input and the list of recipients from the command's arguments. This option is assumed if no other conflicting options are present, so it is possible to inject a message by a simple command, such as:

```
exim theodora@byzantium.example
message
.
```

where *`message`* contains all the necessary RFC 822 header lines, though Exim does add certain headers if they are missing. By default, the message is terminated either by end-of-file (which can be signalled from a terminal by typing CTRL-D) or by a line containing only a single dot, as shown previously. The second form of termination is turned off if *-i* or *-oi* is present, and this should be used whenever messages of unknown content are submitted by this means. Otherwise, a line in the message that consists of a single dot causes the message to be truncated.

The *-t* option provides an alternative way of supplying the message's envelope recipients. If it is present, it implies *-bm* and the recipients are taken from the *To:*, *Cc:*, and *Bcc:* header lines instead of from the command arguments. Any *Bcc:* header lines are then removed from the message. For example:

```
exim -t
From: caesar@rome.example
To: theodora@byzantium.example
Bcc: cleopatra@cairo.example
...
```

submits a message to be delivered to *theodora@byzantium.example* and to *cleopatra@cairo.example*; neither copy will contain the *Bcc:* line. This is the only circumstance in which Exim removes *Bcc:* lines. If they are present in messages received through other interfaces, they are left intact.

If addresses are supplied as command arguments when *-t* is used, there are two possibilities: they can be added to or subtracted from the list of addresses obtained from the header lines. By default, Exim follows the behavior that is documented for many versions of Sendmail and subtracts them from the list. Furthermore, if any of the other addresses subsequently generate one of the argument addresses as a result of aliasing or forwarding, it is also discarded. For example:

```
exim -t brutus@rome.example
From: caesar@rome.example
To: anthony@rome.example
Cc: senate-list@rome.example
...
```

submits a message that is not delivered to *brutus@rome.example*, even if that address appears in the expansion of *senate-list*. Of course, this works only if aliases are expanded on the same host; if the message is dispatched to another host with the address *senate-list@rome.example* intact, the exception is lost. The feature therefore seems to be of little use.

In practice, a number of versions of Sendmail do not follow the documentation. Instead, they add argument addresses to the recipients list. Exim can be made to behave in this way by setting:

```
extract_addresses_remove_arguments = false
```

in its configuration file. When this is done, argument addresses that do not also appear in *To:* or *Cc:* headers behave like additional *Bcc:* recipients.

Exim expects that messages submitted using *-bm* or *-t* contain lines terminated by a single linefeed character, according to the normal Unix convention, and most user agents that use the interface conform to this usage. However, there are some programs that supply lines terminated by two characters, carriage return and line-feed (CRLF), as if in an SMTP session. To cope with these maverick cases, Exim supports the *-dropcr* option. When this is set, all carriage-return characters in the input are dropped.

The *-or* option can be used to set a timeout for receiving a non-SMTP message; this overrides the `accept_timeout` configuration option. If no timeout is set, Exim waits forever for data on the standard input.

Summary of Reception Options

The options used to control the way in which reception processes operate are summarized in Table 20-1.

Table 20-1. Input Mode Options

Option	Meaning
-bd	Start listening daemon
-bs	SMTP on stdin and stdout, from local process or via `inetd`
-bS	Batch SMTP from local process
-bm	Message on standard input, recipients as arguments
-t	Message on standard input, recipients from header lines
-dropcr	Drop carriage-return characters
-r	Set non-SMTP timeout

Additional Message Data

Several options provide additional data to be incorporated into a message received from a local process (that is, not over TCP/IP).

Sender Address

The *-f* option supplies a sender address to override the address computed from the caller's login name, but only if the caller is a trusted user. For example, on a host whose default mail domain is *elysium.example*, if a *root* process obeys:

```
    exim -f zeus@olympus.example apollo@olympus.example
```

the envelope sender of the message is set to *zeus@olympus.example* instead of *root@elysium.example* because *root* is always a trusted user. The *-f* option is ignored by default for nontrusted users. However, from Exim release 3.20, untrusted callers can be allowed to use *-f* by setting the `untrusted_set_sender` option true.

Even when this option is not set, untrusted callers are always permitted to use one special form of *-f*. A call of the form:

```
exim -f '<>' apollo@olympus.example
```

specifies an empty envelope sender for the message.* Empty envelope senders are used as a way of identifying messages that must never give rise to bounce messages. This usage is prescribed in the RFCs for bounce messages themselves, and it has also been adopted for other kinds of messages, such as delivery delay warnings. If a nontrusted user calls Exim in this way, the *-f* option is honored in that the envelope sender is emptied, but unless `no_local_from_check` is set, there is still a comparison of the real sender with the contents of the *From:* header, and a *Sender:* header is added if necessary. This does not happen if the caller is trusted.

If a *-f* option is present on the command line, it overrides any sender information obtained from an initial `From` line at the start of the message.

Sender Name

When Exim constructs a *Sender:* header, or a *From:* header (which it does if one is missing) for a local sender, it reads the system's password information to obtain the caller's real name from the so-called "gecos" field, leading to lines of the form:

```
Sender: The Boss <zeus@olympus.example>
```

The username part of this (`The Boss`) can be overridden by means of the *-F* command-line option, for example:

```
exim -F 'The Big Cheese' apollo@olympus.example
```

Because users are normally permitted to change the values of their gecos fields in the password information, this option is not restricted to trusted users.

Remote Host Information

There are a number of options starting with *-oM* that trusted users can use when submitting a message locally to set values that are normally obtained from an incoming SMTP call. The message then has the characteristics of one that was received from a remote host. These are as follows:

- *-oMa* sets the field that contains the IP address of the remote host.
- *-oMi* sets the field that holds the IP address of the interface on the local host, which was used to receive the message.

* The angle brackets <> are quoted to make this a valid shell command line.

- *-oMr* sets the protocol used to receive the message. This value is useful only for logging; there is no restriction on what it may contain.
- *-oMs* sets the field that holds the verified name of the remote host.
- *-oMt* sets the field that holds the identification string obtained by an RFC 1413 (ident) callback to the sending host.

If any of these options are set by a nontrusted caller, or for SMTP input over TCP/IP, they are ignored, except that a nontrusted caller is permitted to use them in conjunction with the *-bh*, *-bf*, and *-bF* options, for testing host checks and filter files.

Apart from testing, the *-oM* options are useful when submitting mail received from remote hosts by some non-SMTP protocol. Suppose a batch of mail has been received by UUCP from the host *fleeting.example*, whose IP address is 192.168.23.45, and stored in batch SMTP format in a file called */etc/uucp/received*. This could be passed to Exim by a trusted user running the command:

```
exim -bS -oMa 192.168.23.45 -oMs fleeting.example \
  -oMr uucp < /etc/uucp/received
```

The log entries and the *Received:* header line that is added to every message would show the values supplied by the *-oM* options.

Immediate Delivery Control

This set of options controls what happens to a message immediately after it has been received; specifically, whether a delivery process is started for it or not. They are rarely needed except for testing.

The *-odb* option applies to all modes in which Exim accepts incoming messages, including the listening daemon. It requests "background" delivery of such messages, which means that the accepting process automatically starts a delivery process for each message received, but Exim does not wait for such processes to complete (they carry on running "in the background"). This is the default action if none of the *-od* options are present.

The *-odf* and *-odi* options, which are synonymous,* request "foreground" (synchronous) delivery when Exim has accepted a locally generated message.† For a single message received on the standard input, if the protection regime permits it,

* *-odf* is compatible with Smail 3; *-odi* is compatible with Sendmail.

† If given for a daemon process, these are same as *-odb*.

Exim converts the reception process into a delivery process. In other cases, it creates a new delivery process, but waits for it to complete before proceeding. The effect is that the original reception process does not finish until the delivery attempt does.

The *-odq* option applies to all modes in which Exim accepts incoming messages, including the listening daemon. It specifies that the accepting process should not automatically start a delivery attempt for each message received. Messages are placed on the queue and remain there until a subsequent queue runner process encounters them. The `queue_only` configuration option has the same effect.

There are two variations on *-odq* that cause partial delivery of incoming messages. *-odqr* causes Exim to process local addresses when a message is received, but not even to try routing remote addresses. The remote addresses are picked up by the next queue runner. The `queue_remote_domains` configuration option has the same effect for specific domains.

In contrast, if *-odqs* is set, the addresses are all processed and local deliveries are done in the normal way. However, if any SMTP deliveries are required, they are not done at this time. Such messages remain on the queue until a subsequent queue runner process encounters them. Because routing was done, Exim knows which messages are waiting for which hosts, so if there are a number of messages destined for the same host, they are sent in a single SMTP connection. The `queue_smtp_domains` configuration option has the same effect for specific domains. See also the *-qq* option in the section "Two-Pass Processing for Remote Addresses" later in this chapter.

Error Routing

If Exim detects an error while receiving a non-SMTP message (for example, a malformed recipient address), it can report the problem either by writing a message on the standard error file or by sending a mail message to the sender. Which of these two actions it takes is controlled by the following options:

- If *-oem* is set, the error is reported by sending a message. The return code from Exim is 2, if the error was that the original message had no recipients, or 1 otherwise. This is the default action if none of these options are given.
- If *-oee* is set, the error is again reported by sending a message, but this time the return code from Exim is zero if the error message was successfully sent. If sending an error message fails, the return code is as for *-oem*.
- If *-oep* is set, the error is reported by writing a message to the standard error stream and given a return code of 1.

Errors are handled in a special way for batch SMTP input; this is described in the section "Batched SMTP" in Chapter 15.

If there is a problem with the sender address, which can only happen when it is supplied via the *-f* option, an error message is written to the standard error stream, independently of the setting of these options.

Queue Runner Processes

Queue runner processes are normally started periodically by the daemon, or by a *cron* job if you are not using a daemon. They can also be started manually by an admin user, if necessary. For example, the command:

```
exim -q
```

creates a single queue runner process that scans the queue once. This is the command that a daemon issues whenever it is time to start a queue runner.

Overriding Retry Times and Freezing

A normal queue runner processes only unfrozen messages and only addresses whose retry times have been reached. Additional letters can be added to *-q* to change this. If a single `f` follows *-q*, delivery attempts are forced for all addresses (whether or not they have reached their retry times), but frozen messages are still skipped. However, if `ff` follows *-q*, frozen messages are automatically thawed and included in the processing. Thus:

```
exim -qff
```

ensures that a delivery attempt is made for every address in every message.

Local Addresses Only

A queue runner can be restricted to local addresses only (those that match `local_domains`) by adding the letter `l` (ell), following `f` or `ff`, if present. Thus:

```
exim -qfl
```

processes all local addresses in all unfrozen messages, but ignores all remote addresses.

Two-Pass Processing for Remote Addresses

In a conventional queue run, each message is processed only once. If a number of messages have remote addresses that route to the same host and none of them have previously been processed, each is sent in a separate SMTP connection. This circumstance is quite common in some configurations, such as a host that is connected to the Internet only intermittently.*

For better performance, Exim should know that it has several messages for the same host so they can be sent in a single SMTP connection. If any of the *-q* options is specified with an additional `q` (for example, *-qqff*), the resulting queue run is done in two stages. In the first stage, remote addresses are routed, but no transportation is done. The database that remembers which messages are waiting for specific hosts is updated, as if delivery to those hosts had been deferred. When this is complete, a second, normal queue scan happens, and normal directing, routing, and delivery takes place. Messages that are routed to the same host are delivered down a single SMTP connection because of the hints that were set up during the first queue scan.

Periodic Queue Runs

On most installations, queue runner processes should be started at regular intervals. You can request that an Exim daemon do this job by following *-q* with a time value. For example:

```
exim -q20m
```

creates a daemon that starts a queue runner process every 20 minutes (and does nothing else). This form of the *-q* option is usually combined with *-bd* in order to start a single daemon that listens for incoming SMTP as well as periodically starting queue runners. For example:

```
exim -bd -q30m
```

In practice, the command that starts this kind of daemon is usually the standard one that appears in the operating system's boot scripts, which refer to */usr/sbin/sendmail* or */usr/lib/sendmail*. Usually, such scripts are able to start up Exim instead, without needing modification, provided that the Sendmail path has been symbolically symbolically to the Exim binary.

You can, if you wish, use a time value with any of the variants of *-q* discussed so far, for example:

```
exim -qff4h
```

forces a delivery attempt of every address on the queue every four hours.

* See the section "Exim on an Intermittently Connected Host" in Chapter 12, *Delivery Errors and Retrying*, for a detailed discussion of this case.

Processing Specific Messages

By default, a queue runner process scans the entire queue of messages in an unpredicatable order. Sometimes you may want to do a queue run that looks at only the most recently arrived messages on the queue. For example, you might learn that a host that has been offline for a few hours is now working again, but you also know that the older messages are for a host that is still down.

If you follow *-q* (or *-qf*, and so on) with a message ID, all messages whose IDs are lexically less are skipped. For example:

```
exim -qf 0t5C6f-0000c8-00
```

Because message IDs start with the time of arrival, this skips any messages that arrived before `0t5C6f-0000c8-00`. If a second message ID is given, messages whose IDs are greater than it are skipped. However, the queue is still processed in an arbitrary order.

Processing Specific Addresses

A queue runner process can be instructed to process only messages whose senders or recipients match a particular pattern. The *-S* and *-R* options specify patterns for the sender and recipients, respectively. If both *-S* and *-R* are specified, both must be satisfied. In the case of *-R*, a message is selected as long as at least one of its undelivered recipients matches. For example:

```
exim -R zalamea.example
```

starts a delivery process for any message with an undelivered address that contains *zalamea.example*. It is a straightforward textual check; the string may be found in the local part or in the domain (or in both, if it contains an `@`).

If you want to use a more complicated pattern, you can specify that the string you supply is a regular expression, by following *-R* or *-S* by the letter `r`. For example:

```
exim -Rr '(major|minor)\.zalamea\.example$'
```

selects messages that contain an undelivered address that ends with `major.zalamea.example` or `minor.zalamea.example`.

Once a message is selected for delivery, a normal delivery process is started and all the recipients are processed, not just those that matched *-R*. For the first selected message, Exim overrides any retry information and forces a delivery attempt for each undelivered address. If *-S* or *-R* is followed by `f` or `ff`, the forcing applies to all selected messages; in the case of `ff`, frozen messages are also included.

The *-R* option makes it straightforward to initiate delivery of all messages to a given domain after a host has been down for some time.

Summary of Queue Runner Options

The options for queue runner processes are summarized in Table 20-2.

Table 20-2. Queue Runner Options

Option	Meaning
-q	Normal queue runner
-qf	Queue run with forced deliveries
-qff	Forced deliveries and frozen messages
-ql	Local domains only
-qfl	Forced deliveries, local domains only
-qffl	As *-qfl*, but include frozen messages
-R	Select on recipient, literal string
-Rf	Ditto, with forcing
-Rff	Ditto, with forcing and frozen messages
-Rr	Select on recipient, regular expression
-Rrf	Ditto, with forcing
-Rrff	Ditto, with forcing and frozen messages
-S	Select on sender, literal string
-Sf	Ditto, with forcing
-Sff	Ditto, with forcing and frozen messages
-Sr	Select on sender, regular expression
-Srf	Ditto, with forcing
-Srff	Ditto, with forcing and frozen messages

Any of the *-q . . .* options can be given as *-qq . . .* to cause a two-stage queue run, and any of them may also be followed by a time to set up a daemon that periodically repeats the queue run with the same option.

Configuration Overrides

The name of Exim's runtime configuration file is defined in the build-time configuration and embedded in the binary. This is necessary because Exim is normally a setuid program with *root* privilege that can be called by any process. Allowing use of arbitrary runtime configurations would be a huge security exposure. However, it is sometimes useful to be able to use an alternative configuration file or to vary the contents of the standard file, either for testing or for some special purpose.

The runtime configuration file's name can be changed by means of the *-C* option:

```
exim -C /etc/exim/alt.config ...
```

If the caller is not *root* or the Exim user and the filename is different to the built-in name, Exim immediately gives up its *root* privilege permanently and runs as the calling user.

The runtime configuration file can contain macro definitions.* Their values can be overridden by means of the *-D* option, for example:

```
exim -DLOG_LEVEL=6 ...
```

but again, unless the caller is *root* or the Exim user, Exim gives up its root privilege when this option is used. The *-D* option can be repeated up to ten times in a command line.

Watching Exim's Queue

Admin users can use this next set of options to inspect the contents of Exim's queue and the contents of individual messages. If you have access to an X Window system server, an alternative way of looking at this information is to run the Exim monitor (*eximon*). However, anything that *eximon* can do can also be done from the command line.

An admin user can obtain a listing of all the messages currently in the queue by running:

```
exim -bp
```

If this option is used by a nonadmin user, only those messages submitted by the caller are shown. Each message is displayed as in this example:

```
  25m  2.9K  0t5C6f-0000c8-00 <caesar@rome.example>
             brutus@rome.example
             ...
```

The first line shows the length of time the message has been on the queue (in this case, 25 minutes), the size (2.9KB), the local ID, and the envelope sender. For bounce messages that have no sender, <> appears. The remaining lines contain the envelope recipients, one per line. Those to whom the message has already been delivered are marked with the letter `D`; in the case of an address that is expanded by aliasing or forwarding, this happens only when deliveries to all its children are complete.

If *-bpu* is used instead of *-bp*, only undelivered addresses are shown. If *-bpa* is used, delivered addresses that were generated from the original addresses are

* See the section "Macros in the Configuration File" in Chapter 4, *Exim Operations Overview.*

added. If *-bpr* is used, the output is not sorted into chronological order of message arrival. This can speed things up if there are lots of messages on the queue, and is particularly useful if the output is going to be postprocessed in a way that does not require sorting. You can also use *-bpra* and *-bpru*, which act like *-bpa* and *-bpu*, but again without sorting.

Message Control

There is a set of options, all beginning with *-M*, that permit admin users to inspect the contents of messages that are on the queue and perform certain actions on them.

Operations on a List of Messages

The options described in this section can all take a list of message IDs as arguments; the action is performed on each message that is not in the process of being delivered. For example:

```
exim -M 123H3N-0003mY-00 0t5C6f-0000c8-00
```

-M creates a delivery process for each message in turn, thawing it first, if necessary. During delivery, retry times are overridden and options such as `queue_remote_domains` and `hold_domains` are ignored. In other words, Exim carries out a delivery attempt for every undelivered recipient. This is often called "forcing message delivery."

-Mf and *-Mt* freeze and thaw messages, respectively. When *-Mt* has been applied to a message, the condition `manually_thawed` is true in an Exim filter.

-Mg and *-Mrm* both cause messages to be abandoned. The difference between them is that *-Mg* (`g` stands for "give up") fails each address with the error "delivery cancelled by administrator" and generates a bounce message to the sender, whereas *-Mrm* just removes messages from the spool without sending bounces.

If a delivery process is already working on a message, none of these options has any effect and an error message is output. Actions other than *-M* are logged in the main log, along with the identity of the admin user who requested them.

Inspecting a Queued Message

The contents of a message's spool files can be inspected by an admin user; the options *-Mvb*, *-Mvh*, and *-Mvl*, followed by a message ID, output the body (*-D* file), header (*-H* file), or message log file, respectively. For example:

```
exim -Mvl 123H3N-0003mY-00
```

shows the message log for message `123H3N-0003mY-00`.

Modifying a Queued Message

The options described in this section allow an admin user to modify a message on the queue, provided that it is not in the process of being delivered. They all require a message ID as their first argument; some of them need additional data as well.

The recipients of a message can be changed by *-Mar*, *-Mmd*, and *-Mmad*. The first of these adds an additional recipient. For example:

```
exim -Mar 123H3N-0003mY-00 extra@xyz.example
```

adds the recipient *extra@xyz.example* to message `123H3N-0003mY-00`. There is no way to remove recipients, but Exim can be told to pretend that it has delivered to them. The command:

```
exim -Mmad 123H3N-0003mY-00
```

marks all recipient addresses as delivered, whereas:

```
exim -Mmd 123H3N-0003mY-00 godot@waiting.example
```

marks just the address *godot@waiting.example* as delivered. If you need to divert a message to one or more new recipients, perhaps because the original addresses are known to be invalid, the safe way to do it is by:

- Freezing the message using *-Mf* to ensure that no Exim process tries to deliver it while you are working on it.
- Using *-Mmad* to mark all the existing recipients as delivered, or using *-Mmd* to do that to certain recipients only.
- Using *-Mar* as many times as necessary to add new recipients.
- Thawing the message using *-Mt*, or forcing a delivery with *-M*.

You can also change the envelope sender of a message using the *-Mes* option. For example:

```
exim -Mes 123H3N-0003mY-00 newsender@new.domain.example
```

To remove the sender altogether (that is, to make the message look like a bounce message), the new sender may be specified as <>.

Finally, you can edit the body of a message with the *-Meb* option, which takes just a single message ID as its argument:

```
exim -Meb 123H3N-0003mY-00
```

This runs, under */bin/sh*, the command defined in the environment variable `VISUAL` or, if that is not defined, `EDITOR` or, if that is not defined, the command *vi*, on a copy of the spool file containing the body of the message. If the editor exits normally, the result of editing replaces the spool file. The message is locked during this process, so no delivery attempts can occur. Note that the first line of the spool file is its own name; care should be taken not to disturb this.

The original thinking behind providing this feature is that an administrator who has had to mess around with the addresses to get a message delivered might want to add some comment at the start of the message text. However, when messages are in MIME format or have digital signatures, this is not an appropriate thing to do. It is better to arrange for the message to be delivered to yourself and then forward it with a covering note.

Summary of Message Control Options

The options for message control are summarized in Table 20-3.

Table 20-3. Queue Runner Options

Option	Meaning
-M	Force delivery
-Mar	Add recipient
-Meb	Edit body
-Mes	Edit sender
-Mf	Freeze
-Mg	Give up (bounce)
-Mmad	Mark all delivered
-Mmd	Mark delivered
-Mrm	Remove message (no bounce)
-Mt	Thaw
-Mvb	View message body
-Mvh	View message header
-Mvl	View message log

Testing Options

A number of options to help you test out Exim and its configuration or find out why it did what it did. Sometimes the options in the next section, the section "Options for Debugging," can be helpful too.

Testing the Configuration Settings

If you want to be sure that the Exim binary is usable and that it can successfully read its configuration file, run:

```
exim -bV
```

Exim writes its version number, compilation number, and compilation date to the standard output, reads its configuration file, and exits successfully if no problems are encountered. Errors in the configuration file cause messages to be written to the standard error stream.

You can also check what it has read from the configuration file. If *-bP* is given with no arguments, it causes the values of all Exim's main configuration options to be written to the standard output. However, if any of the option settings are preceded by the word `hide`, their values are shown only to admin users. You should use `hide` when you place sensitive information, such as passwords for access to databases, in the configuration file.

The values of one or more specific options can be requested by giving their names as arguments, for example:

```
exim -bP qualify_domain local_domains
```

The name of the configuration file can be requested by:

```
exim -bP configure_file
```

Configuration settings for individual drivers can be obtained by specifying one of the words `director`, `router`, `transport`, or `authenticator`, followed by the name of an appropriate driver instance. For example:

```
exim -bP transport local_delivery
```

The generic driver options are output first, followed by the driver's private options. A complete list of all drivers of a particular type, with their option settings, can be obtained by using `directors`, `routers`, `transports`, or `authenticators`. For example:

```
exim -bP authenticators
```

Finally, a list of the names of drivers of a particular type can be obtained by using one of the words `director_list`, `router_list`, `transport_list`, or `authenticator_list`. For example:

```
exim -bP director_list
```

might output:

```
system_aliases
cancelled_users
real_localuser
userforward
localuser
```

Testing Address Handling

In most common cases, it is not necessary to send a message to find out how Exim would handle a particular address; the *-bt* option does this for you. For example:

```
$ exim -bt ph10@exim.example
ph10@exim.example
  deliver to ph10@exim.example
  router = lookuphost, transport = remote_smtp
  host ppsw.exim.example   [10.111.8.38]    MX=7
  host ppsw.exim.example   [10.111.8.40]    MX=7
```

This tells you that the **lookuphost** router was the one that handled the address, routing it to the **remote_smtp** transport with the given host list. For more information about how this outcome was reached, you can set debugging options (see the section "Options for Debugging" later).

The *-bv* and *-bvs* options allow you to check what Exim would do when verifying a recipient or a sender address, respectively, as opposed to processing the address for delivery. If you have not used options that cause directors or routers to behave differently when verifying (for example, `verify_only`), the result is the same as for *-bt*. There is a difference between *-bv* and *-bvs* only if you have set `verify_sender` or `verify_recipient` on a driver, in order to make a distinction between these two cases.

There is one shortcoming of all these address tests. If you set up a configuration so that the routing or directing process makes use of data from within a message that is being delivered, this cannot be simulated in the absence of a message. For example, suppose you want to send all output from your mailing lists to a central server that has plenty of disk for holding a large queue (because deliveries to a large list can take some time), but you want to deliver other messages directly to their destinations. You can identify mailing list messages by the fact that they contain the header line:

```
Precedence: list
```

Configuring Exim to do this is straightforward, using a router of this kind:

```
list_to_server:
  driver = domainlist
  condition = ${if eq {$h_precedence:}{list}{yes}{no}}
  transport = remote_smtp
  route_list = * server.name.example byname
```

However, when you use *-bt* to test addresses, this router is never run because the condition never matches. See the description of *-N* in the section "Suppressing Delivery," later in this chapter, for an alternative approach to testing that could be more useful here.

Testing Incoming Connections

If you set up verification and rejection policies for use when mail is received from other hosts, following through exactly how the checks are going to be applied can sometimes be quite tricky. The *-bh* option is there to help you. Specify an IP address, and Exim runs a fake SMTP session, as if it had received a connection from that address. While it is doing so, it outputs comments about the checks that it is applying, so you can see what is happening. You can go through the entire SMTP dialog if you want to; if you want to check on relay controls, you have to proceed at least as far as the RCPT commands.

Nothing is written to the log files and no data is written to Exim's spool directory. This is all totally fake, and is purely for the purpose of testing. Here is a transcript of part of a testing session, interspersed with comments:

```
$ exim -bh 192.203.178.4
```

Start up a fake SMTP session, as if a call from 192.203.178.4 had been received:

```
**** SMTP testing session as if from host 192.203.178.4
**** Not for real!
```

Exim is reminding you that this is all make-believe:

```
>>> host in host_lookup? yes (end of list)
```

Exim checks to see if the calling host matches anything in the `host_lookup` option. The answer is "yes," but what it has matched is the end of the list. How can this be? The final item in `host_lookup` must have been a negative item (starting with an exclamation mark). When this is the case, reaching the end of the list yields "yes" rather than "no."

```
>>> looking up hostname for 192.203.178.4
>>> IP address lookup yielded dul.crynwr.com
```

Because the IP address matched `host_lookup`, Exim did a reverse DNS lookup to find the hostname, and what it found was *dul.crynwr.com.* This is a test host that is guaranteed to be on the MAPS DUL (dial-up user list), so that people can test their configurations:

```
>>> host in host_reject? no (option unset)
>>> host in host_reject@_recipients? no (option unset)
```

Two more configuration options are checked; both are unset, so the host does not match either of them:

```
>>> host in rbl_hosts? yes (*)
>>> RBL lookup for 4.178.203.192.dialups.mail-abuse.org succeeded
>>> => that means it is black listed ...
>>> See <http://mail-abuse.org/dul/>
LOG: recipients refused from dul.crynwr.com
  [192.203.178.4] (RBL dialups.mail-abuse.org)
```

The host matched rbl_hosts and the item it matched was *. Exim therefore looked it up using the domain from rbl_domains, which in this case was *dialups.mail-abuse.org*. The lookup succeeded, which means that this host is on the blacklist. The reference to the URL are the contents of the DNS TXT record that is associated with the address. The line starting LOG: is what Exim would write to its log file if this were a real SMTP session. It is going to refuse all the recipients in any messages because the host is blacklisted:

```
220 libra.test.example ESMTP Exim 3.22 ...
```

Exim has now finished its preliminary testing, and this is the initial response it would send to the remote host. You now have to play the part of the client by typing SMTP commands, the first of which should be HELO or EHLO:

```
helo dul.crynwr.com
>>> dul.crynwr.com in local_domains? no (end of list)
250 libra.test.example Hello dul.crynwr.com [192.203.178.4]
```

Exim checks the argument of HELO in case the client has erroneously used the local host's name instead of its own, a common mistake. (When this happens, it forces a DNS lookup to get the real name.) After a successful HELO you can try to "send" a message, using the MAIL, RCPT, and DATA commands, and Exim will output comments about the checks it performs on the addresses and give the appropriate responses. You can end the testing session with the QUIT command or by breaking out, using CTRL-C.

Testing Retry Rules

You can check which retry rule will be used for a particular address by means of the *-brt* option, which must be followed by at least one argument. Exim outputs the applicable retry rule. For example:

```
$ exim -brt bach.comp.example
Retry rule: *.comp.example  F,2h,15m; F,4d,30m;
```

The argument can be a complete email address or just a domain name. Another domain name can be given as an optional second argument; if no retry rule is found for the first argument, the second is tried. This ties in with Exim's behavior when looking for retry rules for remote hosts. If no rule is found that matches the host, one that matches the mail domain is sought.

A third optional argument, the name of a specific delivery error, may also be given. So the following:

```
exim -brt host.comp.example comp.example timeout_connect
```

asks the question "Which retry rule will be used if a connection to the host *host.comp.example* times out while attempting to deliver a message with a recipient in the *comp.example* domain?"

Testing Rewriting Rules

The *-brw* option for testing address rewriting rules is described in the section "Testing Rewriting Rules" in Chapter 14, *Rewriting Addresses*.

Testing Filter Files

The *-bf* and *-bF* options for testing user and system filters are described in the section "Testing Filter Files" in Chapter 10, *Message Filtering*.

Testing String Expansion

The *-be* option for testing string expansions is described in the section "Testing String Expansions" in Chapter 17, *String Expansion*.

Options for Debugging

It is helpful, both to its author and to its users, if a complicated program like Exim can output some information about what it is doing to make it easier to track down problems. You can ask Exim to write debugging information to the standard error stream by setting the *-d* option. This can be followed by a number; the higher the number, the more information is output. Without a number, *-d* is the same as *-d1*, and for compatibility, *-v* and *-ov* (for "verbose") are additional synonyms.

The maximum amount of general information is given when *-d9* is set; *-d10* gives, in addition, details of the interpretation of filter files, and *-d11* or higher turns on the debugging options for DNS lookups, causing the DNS resolver to output copies of its queries and the responses it receives from name servers.

The debugging output is designed primarily to help Exim's author track down problems, but a lot of it should be understandable by most administrators. In particular, if you turn on debugging in conjunction with *-bt* or when delivering a message, you can track the flow of control through the various directors and routers as an address is processed. If any SMTP connections are made, the SMTP dialog is shown at all debugging levels.

The *-dm* option causes information about memory allocation and freeing operations to be written to the standard error stream. This information is very much for the Exim internals expert.

Suppressing Delivery

When discussing *-bt* and *-bv* in the section "Testing Address Handling" earlier in this chapter, it was pointed out that they cannot be used to test any address processing that involves the contents of a message. One way this can be tested is to use the *-N* option. This is a debugging option that inhibits delivery of a message at the transport level. It implies at least *-d1*. Exim goes through many of the motions of delivery but does not actually transport the message. Instead, it behaves as if it had successfully done so. However, it does not make any updates to the retry database, and the log entries for deliveries are flagged with `*>` rather than `=>`.

Once *-N* has been used on a message, it can never be delivered normally. If the original delivery is deferred, subsequent delivery attempts are done automatically with *-N*.

Because *-N* throws away mail, only *root* and the Exim user are allowed to use it in conjunction with *-bd*, *-q*, *-R* or *-M*; that is, to "deliver" arbitrary messages in this way. Any other user can use *-N* only when supplying an incoming message, to which it will apply.

Terminating the Options

A pseudo-option consists of two hyphens whose only purpose is to terminate the options, and therefore cause subsequent command-line items to be treated as arguments rather than options, even if they begin with a hyphen. It is possible (though unlikely) for the local part of an email address to begin with a hyphen; to send a message to such an address, you would need to call Exim like this:

```
    exim -- -oddname@wherever.example
```

Embedded Perl Options

The *-pd* and *-ps* options, which control the way an embedded Perl interpreter is intialized, are described in the section "Running Embedded Perl" in Chapter 17.

Compatibility with Sendmail

Many of Exim's command-line options are directly compatible with Sendmail, for Exim to be installed as a "drop-in" replacement. However, because Exim's design is different, some options are not relevant or operate in a different way. Sendmail also has a number of obsolete options and synonyms that still seem to be used by some older MUAs.

- The following options are recognized by Exim, but do nothing: *-B*, *-h*, *-n*, *-m*, *-om*, *-oo*, and *-x*.
- Any option that begins with *-e* is treated as a synonym for the corresponding option that begins *-oe*.
- *-oeq* is synonymous with *-oep* and *-oew* is synonymous with *-oem*.
- *-i* and *-oitrue* are synonyms for *-oi*.
- *-r* is a synonym for *-f*.
- *-qR* and *-qS* are synonyms for *-R* and *-S*, respectively.

Sendmail interprets a call with the *-bi* option as a request to rebuild its alias file, and there is often a command called *newaliases* whose action is something like "rebuild the data base for the mail aliases file."

Exim does not have the concept of *the* alias file; you can configure as many **aliasfile** directors as you like, though in practice just one is most common. "Rebuilding" may be relevant if you are using a DBM or cdb file for aliases, but not if you are using NIS or a database such as MySQL.

Scripts that are run when the system boots (or at other times) may call *newaliases* or */usr/sbin/sendmail* with the *-bi* option. Exim does nothing if called with *-bi*, unless you specify:

```
bi_command = /the/path/to/some/command
```

in which case it runs the given command, under the uid and gid of the caller of Exim. The value of `bi_command` is just a command name with no arguments. If an argument is required, it must be given by the *-oA* option on the command line.

Calling Exim by Different Names

There are some fairly common command names that are used by other MTAs for performing various mail actions. If you are running Exim, the actions can be requested by means of command-line options, and it would be possible to use shell scripts to set this up. There is an easier way though. If you call the Exim binary under certain other names, by means of symbolic links, it assumes specific options. The supported names are:

mailq:
This name assumes the *-bp* option, which causes Exim to list the contents of the queue.

rsmtp:
This name assumes the *-bS* option, which causes Exim to read batch SMTP from the standard input (this is for Smail compatibility).

rmail:

This name assumes the *-i* and *-oee* options; the former turns off the recognition of a single dot as a message terminator, and the latter changes the handling of errors that are detected on input. The name *rmail* is used by some UUCP systems.

runq:

This name assumes *-q* and causes a single queue run (this is for Smail compatibility).

None of these alternative names are set up by the standard installation scripts. If you want to use them, you must create the symbolic links yourself.

21 *Administering Exim*

Once Exim is up and running, there are a few things that must be done regularly to ensure that it keeps on handling your mail the way you want it to. How much regular attention it needs very much depends on the nature of your installation and the volume of mail you are handling.

One thing you might want to do is to watch what Exim is actually in the process of doing or what it has just done. You can check up on Exim processes using the *exiwhat* utility and you can read the log files directly, or use the Exim monitor to display a rolling main log. A utility script called *exigrep* provides a packaged way of extracting log entries for messages that match a given pattern.

Log files can become very large; normally they are "cycled" on a regular (often daily) basis so that logs for previous days can be compressed. Some operating systems have standard procedures for cycling log files; for those that do not, a utility script called *exicyclog* is provided.

In this chapter, we describe Exim's logging mechanism, the format of the entries that are written when messages are received or delivered, and the options you can use to control what is logged. The available utilities for extracting and displaying log information are also described. After that, the facilities for finding out what Exim processes are doing are covered, including use of the Exim monitor, which is an X11 application for Exim administration. Finally, there is a discussion of the maintenance needs of alias and other datafiles, hints databases, and user mailboxes.

Log Files

Exim writes three different logs, referred to as the *main*, *reject*, and *panic* logs.

- The main log records the arrival of each message and each delivery in a single logical line in each case. The format is as compact as possible in an attempt to keep down the size of log files. Two-character flag sequences make it easy to pick out these lines. A number of other events are also recorded in the main log, some of which are conditional on the setting of configuration options.
- The reject log records information from messages that are rejected as a result of a configuration option (that is, for policy reasons). If the message's header has been read, its contents are written to this log, following a copy of the one-line message that is also written to the main log.
- An entry is written to the panic log when Exim suffers a disastrous error (such as a syntax error in its configuration file). It often (but not always) bombs out afterwards. When all is going well, the panic log should be empty. You should check its contents regularly to pick up any problems.* If Exim cannot open a panic log file, it tries as a last resort to write to the system log (*syslog*).

In addition to these three log files, Exim writes a log file for each message it handles. The names of these per-message logs are the message IDs, and they are kept in the *msglog* subdirectory of the spool directory. The contents of a message log are, in effect, a copy of the main log entries for the message in question. These files are written purely to make it easy for the administrator to find the history of what has happened to a particular message. Unless `preserve_message_logs` is set, a message log is deleted when the message to which it refers is complete.

Log Destination Control

Exim's logs may be written to local files, to *syslog*, or to both. However, it should be noted that many *syslog* implementations use UDP as a transport. They are therefore unreliable in the sense that messages are not guaranteed to arrive at the log host, nor is the ordering of messages necessarily maintained.†

* For example, you could set up a *cron* job to mail you if it finds a nonempty panic log.

† It has also been reported that on large log files (tens of megabytes), you may need to tweak *syslog* to prevent it from syncing the file with each write; on Linux, this has been seen to make *syslog* take over 90 percent of CPU time.

The destination for Exim's logs can be configured when the binary is built, or by setting `log_file_path` in the runtime configuration. This latter string is expanded so it can contain, for example, references to the hostname:

```
log_file_path = /var/log/$primary_hostname/exim_%slog
```

It is generally advisable, however, for the log destination to be included in the binary rather than setting it at runtime because then the setting is available right from the start of Exim's execution. Otherwise, if there is something it wants to log before it has read the configuration file (for example, an error in the configuration file), it will not use the path you want, and may not be able to log at all.

The value of `log_file_path` is a colon-separated list, currently limited to two items at most.* If an item is `syslog`, then *syslog* is used; otherwise the item must either be an absolute path, containing `%s` at the point where `main`, `reject`, or `panic` is to be inserted, or be empty, implying the use of the default path (which is `log/%slog` in the spool directory). The default path is used if nothing is specified. Here are some examples of possible settings:

```
log_file_path=/usr/log/exim_%s
log_file_path=syslog
log_file_path=:syslog
log_file_path=syslog : /usr/log/exim_%s
```

Log data is written only to files for the first of these settings, and only to *syslog* for the second. The third setting uses the default path and *syslog*, and the fourth uses *syslog* and a file path. If there is more than one path in the list, the first is used and a panic error is logged.

Logging to syslog

The use of *syslog* does not change what Exim logs or the format of its messages. The same strings are written to *syslog* as to log files. The *syslog* "facility" is set to `LOG_MAIL` and the program name to `exim`. On systems that permit it (all except ULTRIX), the `LOG_PID` flag is set so that the *syslog* call adds the pid as well as the time and hostname to each line. The three log streams are mapped onto *syslog* priorities as follows:

1. The main log is mapped to `LOG_INFO`.
2. The reject log is mapped to `LOG_NOTICE`.
3. The panic log is mapped to `LOG_ALERT`.

Many log lines are written to both the main and the reject logs, so there will be duplicates if these are routed by *syslog* to the same place.

* The delimiter for most lists in Exim can be changed from a colon to another character, but this option is an exception. A colon *must* be used.

Exim's log lines can sometimes be very long, and some of its reject log entries contain multiple lines when headers are included. To cope with both these cases, entries written to *syslog* are split into separate *syslog* calls at each internal newline, and also after a maximum of 1,000 characters. To make it easy to reassemble them later, each component of a split entry starts with a string of the form `[n/m]` or `[n\m]`, where *n* is the component number and *m* is the total number of components in the entry. The separator is `/` when the line was split because it was too long; if it was split because of an internal newline, the separator is `\`.

For example, if the length limit is 70 instead of 1,000, the following would be the result of a typical rejection message to the main log (LOG_INFO), each line in addition being preceded by the time, hostname, and pid as added by *syslog*:

```
[1/3] 1999-09-16 16:09:43 11RdAL-0006pc-00 rejected from [127.0.0.1] (ph10):
[2/3]  syntax error in 'From' header when scanning for sender: missing or ma
[3/3] lformed local part in "<>" (envelope sender is <ph10@cam.example>)
```

The same error might cause the following lines to be written to the reject log (LOG_NOTICE):

```
[1/14] 1999-09-16 16:09:43 11RdAL-0006pc-00 rejected from [127.0.0.1] (ph10):
[2/14]  syntax error in 'From' header when scanning for sender: missing or ma
[3\14] lformed local part in "<>" (envelope sender is <ph10@cam.example>)
[4\14] Recipients: ph10@some.domain.cam.example
[5\14] P Received: from [127.0.0.1] (ident=ph10)
[6\14]        by xxxxx.cam.example with smtp (Exim 3.22 #27)
[7\14]        id 11RdAL-0006pc-00
[8\14]        for ph10@cam.example; Mon, 16 Apr 2001 16:09:43 +0100
[9\14] F From: <>
[10\14]   Subject: this is a test header
[11\14]   X-something: this is another header
[12\14] I Message-Id: <E11RdAL-0006pc-00@xxxxx.cam.example>
[13\14] B Bcc:
[14/14]   Date: Mon, 16 Apr 2001 16:09:43 +0100
```

Log lines that are neither too long nor contain newlines are written to *syslog* without modification, for example:

```
1999-09-16 16:09:47 SMTP connection from [127.0.0.1] closed by QUIT
```

The times added by *syslog* are normally the same as Exim's timestamps (though in a different format and without the year), but can sometimes be different.

Log Level

The `log_level` configuration option controls the amount of data written to the main log. The higher its value, the more is written. Zero sets a minimal level of logging, with higher levels adding information as shown in Table 21-1.

Table 21-1. Log Levels

Level	Information Logged
1	Rejections because of policy readdressing by the system filter
2	Rejections because of message size
3	Verification failures
4	SMTP timeouts
	SMTP connection refusals because too busy
	SMTP unexpected connection loss
	SMTP (dis)connections when `log_smtp_connections` is set
	SMTP syntax errors when `log_smtp_syntax_errors` is set
	Nonimmediate delivery of SMTP messages because of load level or number of connections, and so on
5	Retry time not reached [for any host]
	Spool file locked (i.e. some other process is delivering the message)
	Message is frozen (when skipping it in a queue run)
	Error message sent to ...
6	Invalid `HELO` and `EHLO` arguments (see `helo_verify`)

A log level of 6 currently causes all possible messages to appear, though higher levels may be defined in the future. The default log level is 5, which is on the verbose side. Rejection information is still written to the reject log in all cases.

If the log level is 5 or higher, "retry time not reached" messages are also written to individual message logs. If the log level is 4 or less, they are suppressed after the first delivery attempt.

Other Options Affecting Log Content

The `log_level` mechanism has turned out to be too simple for controlling the various logging requirements that people have. As a result, a number of other options vary the contents of the main log. Some of them affect the contents of log lines that record message arrivals and deliveries; these are described in the section "Logging Message Reception" and the section "Logging Deliveries" later in this chapter. The others are summarized here:

`log_arguments` (Boolean, default = false)
Setting this option causes Exim to write the options and arguments with which it was called to the main log. This is a debugging feature, added to make it easy to find out with what arguments certain MUAs call the MTA. The logging does not happen if Exim has given up *root* privilege because it was called

with the *-C* or *-D* options. This facility cannot log illegal arguments because the arguments are checked before the configuration file is read. The only way to log such cases is to interpose a script between the caller and Exim.*

`log_queue_run_level` (integer, default = 0)
This option specifies the log level for the messages "start queue run" and "end queue run." Setting it higher than the value of `log_level` causes them to be suppressed.

`log_refused_recipients` (Boolean, default = false)
If this option is set, an entry is written in the main and reject logs for each recipient that is refused for policy reasons. Otherwise, cases in which all recipients are to be refused just cause a single log entry for the message.

`log_rewrites` (Boolean, default = false)
This option causes all address rewriting to be logged as an aid to debugging rewriting rules.

`log_smtp_connections` (Boolean, default = false)
This option turns on more verbose logging of incoming SMTP connections at log level 4. This does not apply to batch SMTP, but it does apply to SMTP connections from local processes that use the *-bs* option, including incoming calls using *inetd*. A log line is written whenever a connection is established or closed. If a connection is dropped in the middle of a message, a log line is always written, but otherwise nothing is written at the start and end of SMTP connections unless `log_smtp_connections` is set.

`log_smtp_syntax_errors` (Boolean, default = false)
If this option is set, syntax errors in incoming SMTP commands are logged at level 4. An unrecognized command is treated as a syntax error. For an external connection, the host identity is given; for an internal connection using *-bs*, the sender identification (normally the calling user) is given.

`rbl_log_headers` (Boolean, default = false)
When this option is set, the headers of each message received from a host that matches an RBL domain are written to the reject log. This can occur only if the recipients of the message are not rejected; that is, if the RBL check is configured to warn only.

`rbl_log_rcpt_count` (Boolean, default = false)
When this option is set and `rbl_reject_recipients` is false, the number of `RCPT` commands for each message received from a host that is in the RBL is written to the reject log. This may be greater than the number of valid recipients in the message.

* An example of such a script, called *logargs.sh*, is provided in Exim's utility directory.

There is one final option that affects logging. When Exim includes data from a message within a log entry, it takes care to ensure that unprintable characters are escaped, so as not to mess up the format of the log. For example, if a message contains these lines in its header:

```
Subject: This subject covers
  more than one line
```

and `log_subject` is set, the text that is written to the log is:

```
T="This subject covers\n  more than one line"
```

Characters whose values are greater than 127 (so-called "8-bit" or "top-bit" characters) are by default printed using octal escape sequences. However, if you set `print_topbitchars`, these characters are considered to be printing characters and are sent to the log unmodified.

Format of Main Log Entries

Each entry in the main log is a single line of text. Some of the lines are quite long, but it is done this way to make it easier to parse the lines in programs that analyze the data. Every line starts with a timestamp of the form:

```
2000-06-30 01:07:31
```

See the section "Timestamps" in Chapter 19, *Miscellany*, for a discussion of the time zone used for Exim's timestamps. Log lines that relate to the reception or delivery of messages have a two-character "flag" after the timestamp to make them readily identifiable. The flags are:

`<=`	for an arrival
`=>`	for a successful delivery
`==`	for deferment of delivery till later
`**`	for a delivery failure

In addition, `->` and `*>` are used for some special kinds of delivery, as described later in this chapter.

Logging Message Reception

The arrival of a message that is not received over TCP/IP is logged by a line of the form shown in this example, which is split over several lines here in order to fit it on the page:

```
2000-06-30 00:11:51 137nTL-0005br-00 <= holly@dwarf.example
  U=holly P=local S=811
  id=Pine.SOL.3.96.1000630001852.21797A-100000@dwarf.example
```

The address that immediately follows `<=` is the envelope sender address, after any rewriting rules have been applied and the `U` field records the login name of the process that called Exim to submit the message.

When a message arrives from another host, the `U` field records the RFC 1413 identity of the user that sent the message, if one was received, and an `H` field identifies the sending host:

```
1995-10-31 08:57:53 0tACW1-0005MB-00 <= kryten@dwarf.example
  H=mailer.dwarf.example [192.168.123.123] U=exim
  P=smtp S=5678 id=20000630091558.B12616@dwarf.example
```

The number given in square brackets is the IP address of the host. If there is just a single name in the `H` field, as shown earlier, it has been verified to correspond to the IP address.[*] If the name is in parentheses, it is the name that was quoted by the remote host in the HELO or EHLO command and has not been verified. If verification yielded a different name to that given for HELO or EHLO, the verified name appears first, followed by the HELO or EHLO name in parentheses. In this example, the client did not give its fully qualified name:

```
H=mm272.lucy.example (mm272) [192.168.215.229]
```

Misconfigured hosts (and mail forgers) sometimes put an IP address, with or without brackets, in the HELO or EHLO command, leading to entries in the log containing extracts such as this:

```
H=(10.21.32.43) [192.168.8.34]
H=([10.21.32.43]) [192.168.8.34]
```

Such entries can be confusing. Only the final address in square brackets can be relied on.

For all messages, the `P` field specifies the protocol used to receive the message. This is set to `asmtp` for messages received from hosts that have authenticated themselves using the SMTP AUTH command. In this case, the name of the authenticator that was used is logged with `A` as the field name. If an authenticated identification was set up by the authenticator's `server_set_id` option, this is logged too, separated by a colon from the authenticator name. For example:

```
A=fixed_plain:ph10
```

If you are using a version of Exim that supports encrypted transfers, the cipher that was used for an incoming message is logged with `X` as the field name. For example:

```
X=TLSv1:DES-CBC3-SHA:168
```

[*] This verification occurs only if Exim's configuration requires it to happen.

If you don't want this, set `tls_log_cipher` to false. Nothing is logged by default when Exim requests a certificate from a client, but if you set `tls_log_peerdn`, the Distinguished Name is logged with `DN` as the field name.

The size of the received message is given in bytes by the `S` field. When the message is delivered, headers may be removed or added so that the size of delivered copies of the message may be different to this value (and indeed may be different to one another).

The ID field records the contents of any existing *Message-Id:* header line. If the message does not contain such a line, nothing is logged, but Exim adds one before delivering the message.

A delivery error (bounce) message is shown with the sender address <>, and if it is a locally generated message, this is normally followed by an `R` field, which is a reference to the local identification of the message that caused the error message to be sent. For example:

```
2000-06-30 00:49:27 137o3j-0005mU-00 <= <> R=137o3e-0005mO-00 U=root
  P=local S=1239
```

records the arrival of a bounce message that was provoked by the message with the ID `137o3e-0005mO-00`.

By setting certain configuration options, you can request that additional data be added to the message reception log line. If `log_received_sender` is set, the original sender of a message is added, after the word `from`, and if `log_received_recipients` is set, a list of all the recipients is added, preceded by the word `for`. This happens after any unqualified addresses are qualified, but before any rewriting (except SMTP-time rewriting) is done. If `log_subject` is set, the contents of the *Subject:* header line are added to the log line, preceded by `T=` (`T` is for "topic" because `S` is already used for "size"). Here is an example in which all this information is requested and the envelope sender is being rewritten:

```
2000-06-30 00:11:51 137nTL-0005br-00 <= holly@dwarf.example
  U=holly P=local S=811
  id=Pine.SOL.3.96.1000630001852.21797A-100000@dwarf.example
  T="Testing subject" from <hc1009@mix.dwarf.example>
  for lister@dwarf.example hal@2001.example
```

Logging Deliveries

Here are examples of delivery log lines for a local and a remote delivery, respectively:

```
1995-10-31 08:59:13 0tACW1-0005MB-00 => marv <marv@hitch.example>
  D=localuser T=local_delivery
1995-10-31 09:00:10 0tACW1-0005MB-00 => monk@holistic.example
  R=lookuphost T=remote_smtp H=holistic.example [192.168.234.234]
```

The D, R, and T fields identify the director or router and the transport that were used for the delivery. The H field identifies the remote host.

When a local delivery is set up by a director, the field that immediately follows => is either just a local part, the name of a file, or a pipe command. This is followed by the original address, in angle brackets, as in the first line in the previous example. If (as a result of aliasing or forwarding) intermediate addresses exist between the original and the final address, the last of these is given in parentheses after the final address. However, log_all_parents can be set to cause all intermediate addresses to be logged.

The generation of a reply message by a filter file gets logged as a "delivery" to the addressee, preceded by >. The D and T items record the director and transport. For example:

```
2000-06-30 09:42:35 137wNf-0000ng-00 => >hermione@hws.thaum.example
  <harry@hws.thaum.example> D=userforward T=address_reply
```

shows that user *harry* has a filter file that used a *reply* command to generate a message to *hermione@hws.thaum.example*. Nearby in the log, often immediately preceding such a line, you will find the entry recording the arrival of the generated message.

When a local delivery occurs as a result of routing rather than directing (for example, messages are being batched up for transmission by some other means), the log entry looks more like that for a remote delivery.

If a shadow transport was run after a successful local delivery, the log line for the successful delivery has an item added on the end, of the form:

```
ST=shadow transport name
```

If the shadow transport did not succeed, the error message is put in parentheses afterwards.

For remote deliveries, if the final delivery address is not the same as the original address (owing to changes made by routers), the original is shown in angle brackets. If log_smtp_confirmation is set, the text in the final response from the remote host (that is, the response when it accepted responsibility for the message) is added to the log line, preceded by C=. A number of MTAs (including Exim) return an identifying string in this response, so logging this information allows messages to be tracked more easily.

When more than one address is included in a single delivery (for example, two SMTP RCPT commands are used in one transaction), the second and subsequent addresses are flagged with -> instead of => so that statistics gathering programs can draw a distinction between copies delivered and addresses delivered. If two or more messages are delivered down a single SMTP connection, an asterisk follows the IP address in the log lines for the second and subsequent messages.

When a delivery is discarded as a result of the command *seen finish* being obeyed in a user's filter file that generates no deliveries, a log entry of the form:

```
1998-12-10 00:50:49 0znuJc-0001UB-00 => discarded
  <low.club@trick4.bridge.example> D=userforward
```

is written, to record why no deliveries are logged for that address. If a system filter discards all deliveries for a message, the log line is:

```
1999-12-14 00:30:42 0znuKe-0001UB-00 => discarded (message_filter)
```

Finally, when the *-N* debugging option is used to prevent deliveries from actually occurring, log entries are flagged with `*>` instead of `=>` or `->`.

Deferred Deliveries

When a delivery is deferred, a line of the following form is logged:

```
1995-12-19 16:20:23 0tRiQz-0002Q5-00 == marvin@endrest.example
  T=smtp defer (146): Connection refused
```

In the case of remote deliveries, the error is the one that occurred for the last IP address that was tried. Details of individual SMTP failures are also written to the log, so the previous line would be preceded by a line such as this:

```
1995-12-19 16:20:23 0tRiQz-0002Q5-00 mail12.endrest.example
  [192.168.0.62]: Connection refused
```

When a delivery is deferred because a retry time has not been reached, a defer message is written to the log, but only if `log_level` is at least 5.

- "Retry time not reached" means that the address previously suffered a temporary error during directing or routing or local delivery and the time to retry it has not yet arrived.
- "Retry time not reached for any host" means that the address previously suffered temporary errors during remote delivery and the retry time has not yet arrived for any of the hosts to which it is routed.

Delivery Failures

If a delivery fails, a line of the following form is logged:

```
1995-12-19 16:20:23 0tRiQz-0002Q5-00 ** jim@trek99.example
  <jim@trek99.example>: unknown mail domain
```

Later in the log, a line gives the address to which the delivery error message has been sent, but only if the log level is 5 or higher. For example:

```
1995-12-19 16:20:25 0tRiQz-0002Q5-00 Error message sent to
  spock@vulcan.example
```

If the message has many recipients, this line might be much further down the log because Exim does not send the bounce message until all the recipients have been processed.

Message Completion

A line of the form:

```
1995-10-31 09:00:11 0tACW1-0005MB-00 Completed
```

is written to the main log when a message is about to be removed from the spool at the end of its processing. This guarantees that no further log entries for `0tACW1-0005MB-00` will be written.

Other Log Entries

Various other types of log entries are written from time to time. Most should be self-explanatory. One that sometimes causes worry is "Spool file is locked." This means that an attempt to deliver a message cannot proceed because some other Exim process is already working on the message. This is not normally an error and it can be quite common if queue runner processes are started at frequent intervals. The message can be suppressed by setting the log level to less than 5.

However, if you see this log line repeating for the same message for an unreasonable amount of time, there may be a problem. You can use the *exiwhat* utility to find out what the Exim process that is working on the message is trying to do.

Cycling Log Files

If you are using local files rather than *syslog* (the most common configuraion) to record Exim's logs, you should normally "cycle" the main log and the reject log periodically. The cycling process consists of renaming the current log file and deleting previous ones that are too old. Most sites do this by setting up a *cron* job that runs once a day, commonly at midnight, so that each file contains the log for one day.*

* You cannot, of course, ensure that the renaming happens on the dot of midnight, nor can you synchronize with any Exim processes that might be in the process of writing to the log, so in practice there will usually be a few log lines in the "wrong" file.

An Exim delivery process opens the main log when it first needs to write to it, and it keeps the file open in case subsequent entries are required: for example, if a number of different deliveries are being done for the same message. However, remote SMTP deliveries can take a long time, and this means that the file might be kept open and used long after it was renamed. To avoid this, Exim checks the main log file by name before reusing an open file, and if the file does not exist, or if its inode has changed (that is, although it has the same name, it is actually a different file), the old file is closed and Exim tries to open the main log again from scratch. Thus, an old log file may remain open for quite some time, but no Exim processes should write to it once it has been renamed.

Some operating systems have standard scripts for log cycling, which of course can be used. For those that do not, a utility script called *exicyclog* is provided as part of the Exim distribution. It cycles both the main log and the reject log files. You can run it from a root *crontab* entry of the form:

```
1 0 * * *  /usr/exim/bin/exicyclog
```

which runs the script at 00:01 every day. However, the script does not need to be run as root if an Exim user is defined, because in that case, the log files are owned by that user. One way to run the script as the Exim user is to use this *crontab* entry:

```
1 0 * * *  su exim -c /usr/exim/bin/exicyclog
```

If no main (or reject) log file exists, the script does nothing (for that set of files). Otherwise, each time *exicyclog* is run, the files get "shuffled down" by one: *mainlog* becomes *mainlog.01*, the previous *mainlog.01* becomes *mainlog.02*, and so on, up to a limit that is set in the script when it is built (the default is 10). All the old files except for yesterday's log (*mainlog.01*) are automatically compressed to save disk space.*

Extracting Information from Log Files

Two Perl scripts used to extract information from main log files are provided in the Exim distribution. They provide fairly basic facilities, but of course you can modify them or write your own if you need additional functionality.

* We have used the default filename, *mainlog*, in this description, but the script works fine with whatever log filenames you choose to use.

The exigrep Utility

The `exigrep` utility extracts from one or more log files all entries relevant to any message whose entries contain at least one that matches a given pattern. For example:

```
exigrep 'H=orange\.csi\.example' /var/spool/exim/log/mainlog
```

not only picks out the lines containing the string `H=orange.csi.example`, but also all the other lines for the messages that have an entry that matches. Thus, the output would be the complete set of log lines for all messages involving that particular host. The entries are sorted so that all the entries for each message are printed together, and there is a blank line between each message's entries.

exigrep makes it easy to search for all mail for a given user or a given host or domain. The script's usage is as follows:

```
exigrep [-l] <pattern> [<log file>] ...
```

where the *-l* flag means "literal," that is, treat all characters in the pattern as standing for themselves. Otherwise the pattern must be a Perl regular expression. The log files can be compressed or uncompressed; those that are compressed are piped through the *zcat* utility as they are read.* If no filenames are given on the command line, the standard input is read.

The eximstats Utility

A Perl script called *eximstats* is supplied with the Exim distribution. It extracts statistics from Exim log files. Originally, it was intended merely as a demonstration of how this could be done, but it has found its way into regular use. Over time, it has been hacked about quite a bit and it now gives quite a lot of information by default, but there are options for suppressing various parts of it. Following any options, the arguments to the script are a list of files, which should be main log files. For example:

```
eximstats -nr -ne /var/spool/exim/log/mainlog.01
```

eximstats extracts information about the number and volume of messages received from or delivered to various hosts. The information is sorted both by message count and by volume, and the top 50 hosts in each category are listed on the standard output. For messages delivered and received locally, similar statistics are produced per user.

* This assumes that the location of *zcat* was known at the time Exim was built.

The output also includes total counts and statistics about delivery errors and histograms showing the number of messages received and deliveries made in each hour of the day. A delivery with more than one address in its "envelope" (for example, an SMTP transaction with more than one RCPT command) is counted as a single delivery.

Though normally more deliveries than receipts are reported (because messages may have multiple recipients), it is possible for *eximstats* to report more messages received than delivered, even though the spool is empty at the start and end of the period in question. If an incoming message contains no valid recipients, no deliveries are recorded for it. An error report is handled as a separate message.

eximstats outputs a grand total summary giving the volume and number of messages received and deliveries made and the number of hosts involved in each case. It also outputs the number of messages that were delayed (that is, not completely delivered at the first attempt), and the number that had at least one address that failed. Here is an example of this initial output:

```
Exim statistics from 1999-01-21 00:11:08 to 1999-01-22 00:10:46

Grand total summary
-------------------
                                                At least one address
  TOTAL       Volume    Messages    Hosts      Delayed       Failed
  Received    153MB       16520     2341      53  0.3%    104  0.6%
  Delivered   182MB       23197     1513
```

The remainder of the output is in sections that can be independently disabled or modified by various options. First, there is a summary of deliveries by transport:

```
Deliveries by transport
-----------------------
                      Volume    Messages
  **bypassed**           960           1
  :blackhole:           27KB           4
  address_file        1665KB         425
  address_pipe        2134KB         417
  address_reply         4368           3
  local_delivery       135MB       16141
  remote_smtp           43MB        6206
```

`**bypassed**` is recorded when a message is directed to */dev/null*, which Exim recognizes as a special case. `:blackhole:` records the use of the special `:blackhost:` feature of alias files. This part of the output can be suppressed by setting the *-nt* option.

The next part of the output consists of two textual histograms, showing the number of messages received and delivered per hour, respectively. They are automatically scaled, which is why the numbers of messages per dot in these examples are rather strange:

```
Messages received per hour (each dot is 27 messages)
----------------------------------------------------

00-01    342 ............
01-02    249 .........
02-03    206 .......
03-04    154 .....
04-05    134 ....
05-06    160 .....
06-07    141 .....
07-08    245 .........
08-09    562 ....................
09-10   1208 ............................................
10-11   1228 .............................................
11-12   1300 ................................................
12-13   1242 ..............................................
13-14   1070 .......................................
14-15   1320 ................................................
15-16   1335 .................................................
16-17   1281 ...............................................
17-18   1026 ......................................
18-19    890 ................................
19-20    597 ......................
20-21    452 ................
21-22    448 ................
22-23    467 .................
23-24    463 .................

Deliveries per hour (each dot is 47 deliveries)
-----------------------------------------------

00-01    411 ........
01-02    266 .....
02-03    236 .....
03-04    189 ....
04-05    139 ..
05-06    208 ....
06-07    164 ...
07-08    263 .....
08-09    985 ....................
09-10   1801 ......................................
10-11   1780 .....................................
11-12   1916 ........................................
12-13   1624 ..................................
13-14   1607 ..................................
14-15   2087 ............................................
15-16   2373 ..................................................
16-17   1764 .....................................
17-18   1299 ...........................
18-19   1118 .......................
19-20    693 ..............
20-21    579 ............
21-22    524 ...........
22-23    634 .............
23-24    537 ...........
```

By default, the time interval is one hour. If *-h0* is given, the histograms are suppressed; if *-h* followed by a number is given, the value gives the number of divisions per hour, so *-h2* sets an interval of 30 minutes and the default is equivalent to *-h1*.

Next, there is an analysis of the time spent on the queue, first by all messages:

```
Time spent on the queue: all messages
-------------------------------------

Under   1m    16271  98.5%   98.5%
        5m      168   1.0%   99.5%
       15m       30   0.2%   99.7%
       30m       19   0.1%   99.8%
        1h       10   0.1%   99.9%
        3h       10   0.1%   99.9%
        6h        4   0.0%  100.0%
       12h        3   0.0%  100.0%
        1d        1   0.0%  100.0%
Over    1d        1   0.0%  100.0%
```

and then by messages that had at least one remote delivery:

```
Time spent on the queue: messages with at least one remote delivery
-------------------------------------------------------------------

Under   1m     5073  95.6%   95.6%
        5m      167   3.1%   98.7%
       15m       27   0.5%   99.2%
       30m       12   0.2%   99.5%
        1h       10   0.2%   99.7%
        3h       10   0.2%   99.8%
        6h        4   0.1%   99.9%
       12h        3   0.1%  100.0%
Over    1d        1   0.0%  100.0%
```

These particular statistics were recorded on a very good day. This output can be suppressed by the *-q0* option. Alternatively, *-q* can be followed by a list of time intervals for this analysis. The values are separated by commas and are in seconds, but can involve arithmetic multipliers, so, for example, you can set `3*60` to specify 3 minutes. A setting such as:

```
-q60,5*60,10*60
```

causes *eximstats* to give counts of messages that stayed on the queue for less than one minute, less than 5 minutes, less than 10 minutes, and over 10 minutes.

Unless *-nr* is specified, there follows a list of all messages that were relayed via the local host, starting off like this:

```
Relayed messages
----------------

    5 (rosemary) [192.168.182.138] jcbreb@cus.example
      => green.gra.example [192.168.8.57] r5j4m@herm.gra.example
    ...

Total: 1949 (plus 0 unshown)
```

Each pair of lines represents a single relay route; the leading number shows how many different messages were delivered over this route. The rest of the first line contains the sending hostname and IP address and the envelope sender address.

The relay information lists messages that were actually relayed; that is, they came from a remote host and were delivered to some other remote host directly. A delivery that is considered as a relay by the checking features described in the section "Relay Control" in Chapter 13, *Message Reception and Policy Controls*, because its domain is not in `local_domains`, might still end up being delivered locally under some configurations. If this happens, it does not show up as a relay in the *eximstats* output.

Sometimes you only want to know about certain relays. You can selectively omit relay information by providing a regular expression after *-nr*, like this:

```
eximstats '-nr/busy\.host\.name/' /var/spool/exim/log/mainlog.01
```

The pattern matched against a string of the following form:

```
H=<host> [<ip address>] A=<sender address> => H=<host> A=<recipient address>
```

for example:

```
H=in.host [10.2.3.4] A=from@some.where => H=out.host A=to@else.where
```

The sending hostname appears in parentheses if it has not been verified as matching the IP address. The mail addresses are taken from the envelope, not the headers. Relays that are suppressed by this mechanism contribute to the "unshown" count in the final total line.

The next part of the output consists of "league tables," starting with:

```
Top 50 sending hosts by message count
-------------------------------------

 3178 14592162   local
 1049 14995154   mauve.csi.example
  991 11919355   lilac.csi.example
  990 13009635   navy.csi.example
  711  1418843   red.csi.example
  595  2673643   green.csi.example
  537  3563842   violet.csi.example
  169   419187   (murphy.novore.example)
  101   261352   (imelda.cpug.example)
  ...
```

which lists the hosts from which the most messages were received. It is followed by:

- Top 50 sending hosts by volume (as shown earlier, but based on volume).
- Top 50 local senders by message count; this lists the local parts for messages originating on the local host.
- Top 50 local senders by volume.
- Top 50 destinations by message count; this is by host.
- Top 50 destinations by volume.
- Top 50 local destinations by message count; this is by local part.
- Top 50 local destinations by volume.

You can change the number 50 by means of the *-t* option; for example, *-t10* lists only the top 10 in each category. If you set *-t0*, this part of the output is suppressed; if you set *-tnl*, the information about local senders and destinations is suppressed.

The final section of output from *eximstats* is a list of delivery errors:

```
List of errors
--------------

    1 " peter"@cus.example D=unknownuser T=unknownuser_pipe:
      return message generated

    1 1996@southwest.cim.example R=lookuphost T=smtp: SMTP error
      from remote mailer after RCPT TO: <1996@southwest.cim.example>:
      host tuert.southwest.cim.example [192.168.9.19]:
      550 <1996@southwest.cim.example>...
      User unknown

    ...

Errors encountered: 118
-----------------------
```

The number at the start of each item is a count of the number of identical errors. This output can be suppressed by specifying *-ne*.

Watching What Exim is Doing

There are two aspects to watching what Exim is actually doing at any time. The message files in its spool directory contain the work that it has undertaken to do and represent its activity on a relatively long time scale, whereas the current set of Exim processes are what it is actually doing at this instant. Facilities are provided for looking at both of these kinds of activity.

The exiqsumm Utility

One way of watching what Exim is doing is to inspect the list of messages it is in the process of handling. The *-bp* command-line option and its variants (see the section "Watching Exim's Queue" in Chapter 20, *Command-Line Interface to Exim*) provide this information. There is also a short utility script called *exiqsumm* that postprocesses it to provide a summary, so the command:

```
exim -bp | exiqsumm
```

produces output lines of this form:

```
    3    2322     74m   66m   wek.example
```

This means that three messages on the queue have undelivered addresses in the *wek.example* domain. Their total size is 2322 bytes; the oldest has been queued for 74 minutes and the youngest for 66 minutes.

The exinext Utility

If you want to know when Exim will next try a particular delivery that has suffered a temporary error, you can use a utility called *exinext*, which is mostly a Perl script, to extract information from Exim's retry database. Given a mail domain or a complete address, it looks up the hosts for the domain and outputs any retry information that it may have. At present, the retry information is obtained by running *exim_dumpdb* (see example in this section) and postprocessing the output. *exinext* is not particularly efficient, but then it is not expected to be run very often. For example:

```
$ exinext piglet@milne.fict.example
kanga.milne.fict.example:192.168.8.1 error 146: Connection refused
  first failed: 21-Feb-1996 14:57:34
  last tried:   21-Feb-1996 14:57:34
  next try at:  21-Feb-1996 15:02:34
roo.milne.fict.example:192.168.8.3 error 146: Connection refused
  first failed: 20-Jan-1996 13:12:08
  last tried:   21-Feb-1996 11:42:03
  next try at:  21-Feb-1996 19:42:03
  past final cutoff time
```

The phrase "past final cutoff time" means that an error has been occurring for longer than the maximum time mentioned in the relevant retry rule. You can give *exinext* a local part, without a domain, to obtain retry information for a local delivery that has been failing temporarily. A message ID can be given to obtain retry information pertaining to a specific message. This exists only when an attempt to deliver a message to a remote host suffers a message-specific error (see the section "Message Errors" in Chapter 12, *Delivery Errors and Retrying*).

Querying Exim Processes

You can, of course, use the Unix *ps* command to obtain a list of Exim processes. Typically, this is combined with *grep* to form a command such as:

```
ps -ef | grep exim
```

The output might contain lines such as this:

```
exim   295     1  0   Jul 01 ?        0:05 /usr/sbin/sendmail -bd -q15m
exim  4240   295  0 10:09:40 ?        0:00 /usr/sbin/sendmail -bd -q15m
exim  4342   295  0 10:10:59 ?        0:00 /usr/exim/bin/exim -q
exim  4345  4342  0 10:10:59 ?        0:00 /usr/exim/bin/exim -q
exim  4376   295  0 10:11:13 ?        0:00 /usr/sbin/sendmail -bd -q15m
```

These processes are all running as the Exim user, and none of them has an associated terminal (that's what the question marks mean). You can deduce that process 295 is a daemon process because its parent is process number 1, the *init* process that becomes the parent of all daemon processes. Also, it was started some days ago, so its starting time is given as a date rather than a time.

Processes 4240, 4342, and 4376 are all children of the daemon. The first and the third must be reception processes for incoming SMTP calls because the command that is shown is the same as the daemon's. This means that it has forked new processes, but has not executed a new command. Process 4342, however, although also forked from the daemon, is running a different command (namely, a queue runner process) and has created process 4345 to deliver a message.

This information from *ps* is rather limited. You cannot tell, for example, from which remote hosts messages are arriving, or which messages a queue runner is delivering. A technique that is adopted by some programs for making their activities available is to change the values of the argument variables with which they are called so that the output from *ps* changes as the program proceeds. Unfortunately, this technique does not work on all operating systems, so Exim does not use it.

Instead, Exim processes respond to the `SIGUSR1` signal by writing a line of text describing what they are doing to the file *exim-process.info* in Exim's spool directory. This facility is packaged up for use via a utility script called *exiwhat.* You must run this command as root, so that it has the privilege to send a signal to processes running under any uid.*

* This is different from restarting the daemon using `SIGHUP`, which can be done either as *root* or as *exim.*

The first thing *exiwhat* does is empty the process information file. Then it uses *ps* to find all processes running Exim and sends each one a SIGUSR1 signal. The script waits for one second to allow the Exim processes to react, then copies the file to the standard output. It might look like this:

```
 295 daemon: -q15m, listening on port 25
4240 handling incoming call from [192.168.243.242]
4342 running queue: waiting for 0tAycK-0002ij-00 (4345)
4345 delivering 0tAycK-0002ij-00 to mail.ref.example [192.168.42.42]
  (editor@ref.example)
4375 handling incoming call from [192.168.234.111]
```

The number at the start of each output line is the process number.* The fourth line has been split here, in order to fit it on the page.

Unfortunately, the *ps* command varies between different versions of Unix. Not only are different options used, but the format of the output is different. If it does not seem to work for you, check the first few noncomment lines of the shell script, which should be something like this:

```
ps_cmd=/bin/ps
ps_arg=-e
kill_arg=-USR1
egrep_arg=' exim( |$)'
```

The first line sets the path to the *ps* command and the second is the argument for *ps*. The third sets the argument for the *kill* command to make it send a SIGUSR1 signal and the fourth is the argument for *egrep* to make it extract the list of Exim processes from the *ps* output. The actual values may vary, depending on which operating system you are running.

If you find you need to change these values and you have compiled and installed Exim from the source code, you should change the defaults at compile time so that the right values will be used when you build the next release.

The Exim Monitor

The Exim monitor is an X Window system application that continuously displays information about what Exim is doing. An admin user can perform certain operations on messages from this GUI interface; however, all such facilities are also available from the command line, and, indeed, the monitor itself makes use of the command-line interface to carry out these operations.

* Up to and including Release 3.16, the Exim version number was also given, but this rather redundant information was removed in later versions.

Running the Monitor

The monitor is started by running the script called *eximon*. This is a shell wrapper script that sets up a number of environment variables and then runs the binary called *eximon.bin*. The environment variables are a way of configuring the monitor, for example, specifying the size of its window. Their default values are specified when Exim is built. However, even if you are using a precompiled version of Exim, the parameters that are built into the *eximon* script at compile time can be overridden for a particular invocation by setting up environment variables of the same names, preceded by `EXIMON_`. For example, a shell command such as:

```
EXIMON_LOG_DEPTH=400 eximon
```

(in a Bourne-compatible shell) runs *eximon* with an overriding setting of the `LOG_DEPTH` parameter.

If `EXIMON_LOG_FILE_PATH` is set in the environment, it overrides the Exim log file configuration. This makes it possible to have *eximon* tailing log data that is written to *syslog*, provided that `MAIL.INFO` *syslog* messages are routed to a file on the local host. Otherwise, if only *syslog* is used to record logging data, the Exim monitor is unable to provide a log tail display.

X resources can be used to change the appearance of the window in the normal way. For example, a resource setting of the form:

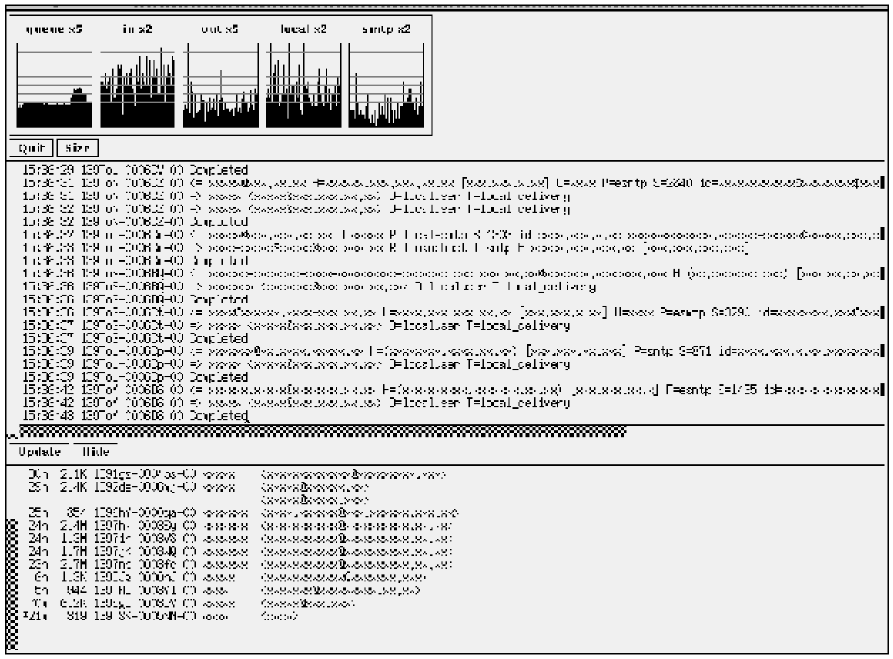
```
Eximon*background: gray94
```

changes the color of the background to light gray rather than white. The stripcharts are drawn with both the data lines and the reference lines in black. This means that the reference lines are not visible when on top of the data. However, their color can be changed by setting a resource called "highlight" (an odd name, but that is what the Athena stripchart widget uses). For example, if your X server is running Unix, you could set up lighter reference lines in the stripcharts by obeying the following:

```
xrdb -merge <<End
Eximon*highlight: gray50
End
```

In order to see the contents of messages on the spool and to operate on them, *eximon* must either be run as *root* or by an admin user; that is, a user who is a member of the Exim group.

The monitor's window is divided into three parts, as shown in Figure 21-1, which is an actual screen shot taken on a live system. However, mail addresses, hostnames, and IP addresses have been hidden with x's for privacy reasons.



```
15:36:29 [illegible] Completed
15:36:31 [illegible] <= [illegible] P=esmtp S=2840 id=[illegible]
15:36:31 [illegible] => [illegible] D=localuser T=local_delivery
15:36:32 [illegible] => [illegible] D=localuser T=local_delivery
15:36:32 [illegible] Completed
[illegible]
[illegible]
[illegible] Completed
[illegible]
[illegible] T=local_delivery
[illegible] Completed
[illegible] <= [illegible] P=esmtp S=3291 id=[illegible]
[illegible] => [illegible] D=localuser T=local_delivery
[illegible] Completed
[illegible] <= [illegible] P=esmtp S=871 id=[illegible]
[illegible] => [illegible] D=localuser T=local_delivery
[illegible] Completed
15:36:42 [illegible] <= [illegible] P=esmtp S=1435 id=[illegible]
15:36:42 [illegible] => [illegible] D=localuser T=local_delivery
15:36:43 [illegible] Completed
```

```
 30h  2.1K [illegible]
 25h  2.4K [illegible]
 25h   854 [illegible]
 24h  2.4M [illegible]
 24h  1.3M [illegible]
 24h  1.7M [illegible]
 23h  2.7M [illegible]
  6h  1.3K [illegible]
  5h   944 [illegible]
 10h  6.2K [illegible]
*21h   819 [illegible]
```

Figure 21-1. Monitor screenshot

The first part contains one or more stripcharts and two action buttons, the second contains a "tail" of the main log file, and the third is a display of the queue of messages awaiting delivery, with two more action buttons.

The Stripcharts

The first stripchart is a count of messages on the queue. The remaining stripcharts are defined in the configuration script by regular expression matches on log file entries, making it possible to display, for example, counts of messages delivered to certain hosts or using certain transports. The supplied defaults display counts of received and delivered messages, and of local and SMTP deliveries. The default period between stripchart updates is one minute.

The stripchart displays rescale themselves automatically as the value they are displaying changes. There are always 10 horizontal lines in each chart; the title string indicates the value of each division when it is greater than one. For example, `x2` means that each division represents a value of 2.

It is also possible to have a stripchart that shows the percentage fullness of a particular disk partition, which is useful when local mailboxes are confined to a single partition. This relies on the availability of the `statvfs()` function or equivalent in the operating system. Most, but not all, versions of Unix that support Exim have this. For this particular stripchart, the top of the chart always represents 100 percent, and the scale is given as `x10%`. You can start up *eximon* with this additional stripchart by a command of this form:

```
EXIMON_SIZE_STRIPCHART=/var/mail eximon
```

assuming you are running a Bourne-compatible shell. This example monitors the size of the partition containing the */var/mail* directory. If you build Exim from source, you can specify in the build-time configuration that this is to be the default. The name of the chart is the last component of the path, but you can change this by setting EXIMON_SIZE_STRIPCHART_NAME if you want to.

Main Action Buttons

Below the stripcharts is an action button for quitting the monitor. Next to this is another button marked `Size`. They are placed here so that shrinking the window to its minimum size leaves just the queue count stripchart and these two buttons visible. The `Size` button is a toggle that causes the window to flip between its maximum and minimum sizes. When expanding to the maximum, if the window cannot be fully seen where it currently is, it is moved back to where it was the last time it was at full size. The old position is remembered and next time the window is reduced to the minimum, it is moved back.

The idea is that you can keep a reduced window just showing one or two stripcharts at a convenient place on your screen, easily expand it to show the full window when required, and just as easily put it back to what it was. This feature is copied from what the *twm* window manager does for its `f.fullzoom` action. The minimum size of the window can be changed by setting the environment variables EXIMON_MIN_HEIGHT and EXIMON_MIN_WIDTH when starting the monitor.

The Log Display

The second section of the window is an area in which a display of the main log tail is maintained. This is not available when the only destination for logging data is *syslog*, unless the syslog lines are routed to a local file whose name is passed to *eximon* via the `EXIMON_LOG_FILE_PATH` environment variable.

The log subwindow has a scrollbar at its lefthand side that can be used to move back to look at earlier text, and the up and down arrow keys also have a scrolling effect. If new text arrives in the window when it is scrolled back, the caret remains where it is, but if the window is not scrolled back, the caret automatically moves to the end of the new text.

The amount of log that is kept depends on the setting of `EXIMON_LOG_BUFFER`, which specifies the amount of memory to use. When this is full, the earlier 50 percent of data is discarded; this is much more efficient than throwing it away line by line. The subwindow also has a horizontal scrollbar for accessing the ends of long log lines. This is the only means of horizontal scrolling; the right and left arrow keys are not available. Text can be cut from this part of the window using the mouse in the normal way. The size of this subwindow is controlled by `EXIMON_LOG_DEPTH`.

Searches of the text in the log window can be carried out by means of the CTRL-R and CTRL-S keystrokes, which default to a reverse and forwards search, respectively. The search covers only the text that is displayed in the window. It cannot go further back up the log. The point from which the search starts is indicated by a caret marker. This is normally at the end of the text in the window, but can be positioned explicitly by pointing and clicking with the left mouse button, and is moved automatically by a successful search.

Pressing CTRL-R or CTRL-S pops up a window into which the search text can be typed. There are buttons for selecting forward or reverse searching, for carrying out the search, and for cancelling. If the `Search` button is pressed, the search happens and the window remains so that further searches can be done. If the Enter key is pressed, a single search is done and the window is closed. If CTRL-C is pressed, the search is cancelled.*

The Queue Display

The bottom section of the monitor window contains a list of all messages that are on the queue, including those currently being delivered, as well as those awaiting delivery in the future.

* The searching facility is implemented using the facilities of the Athena text widget. By default, this pops up a window containing both "search" and "replace" options. In order to suppress the unwanted "replace" portion for *eximon*, a modified version of the `TextPop` widget is distributed with Exim. However, the linkers in BSDI and HP-UX seem unable to handle an externally provided version of `TextPop` when the remaining parts of the text widget come from the standard libraries. On these systems, therefore, *eximon* has to be built with the standard widget, which results in these unwanted items in the search pop-up window.

The depth of this subwindow is controlled by EXIMON_QUEUE_DEPTH, and the frequency with which it is updated is controlled by EXIMON_QUEUE_INTERVAL; the default is 5 minutes because queue scans are quite expensive. However, there is an Update action button just above the display that can be used to force an update of the queue display at any time.

When a host is down for some time, a lot of pending mail can build up for it, and this can make it hard to deal with other messages on the queue. To help with this situation, there is a button next to Update called Hide. If pressed, a dialog box called "Hide addresses ending with" opens. If you type anything in here and press Enter, the text is added to a chain of such texts, and if every undelivered address in a message matches at least one of the texts, the message is not displayed.

If an address does not match any of the texts, all the addresses are displayed as normal. The matching happens on the ends of addresses so, for example, foo.com.example specifies all addresses in that domain, whereas xxx@foo.com.example specifies just one specific address. When any hiding has been set up, a button called Unhide is displayed. If pressed, it cancels all hiding. Also, to ensure that hidden messages are not forgotten, a hide request is automatically cancelled after one hour.

While the dialog box is displayed, you cannot press any buttons or do anything else to the monitor window. For this reason, if you want to cut text from the queue display to use in the dialog box, you must do the cutting before pressing the Hide button.

The queue display contains, for each unhidden queued message, the length of time it has been on the queue, the size of the message, the message ID, the message sender, and the first undelivered recipient, all on one line. If it is a delivery error message, the sender is shown as <>. If there is more than one recipient to which the message has not yet been delivered, subsequent ones are listed on additional lines, up to a maximum configured number, following which an ellipsis is displayed. Recipients that have already received the message are not shown. If a message is frozen, an asterisk is displayed at the lefthand side.

The queue display has a vertical scrollbar and can also be scrolled by means of the arrow keys. Text can be cut from it using the mouse in the normal way. The text searching facilities, as described earlier for the log window, are also available, but the caret is always moved to the end of the text when the queue display is updated.

The Queue Menu

If the Shift key is held down and the left button is clicked when the mouse pointer is over the text for any message in the queue window, an action menu pops up and the first line of the queue display for the message is highlighted. This does not affect any selected text. If you want to use some other event for popping up the menu, you can set `EXIMON_MENU_EVENT` in the environment before starting the monitor. The value set in this parameter is a standard X event description. For example, to run *eximon* using Ctrl rather than Shift you could use:

```
EXIMON_MENU_EVENT='Ctrl<Btn1Down>' eximon
```

An example of an *eximon* menu is shown in Figure 21-2.

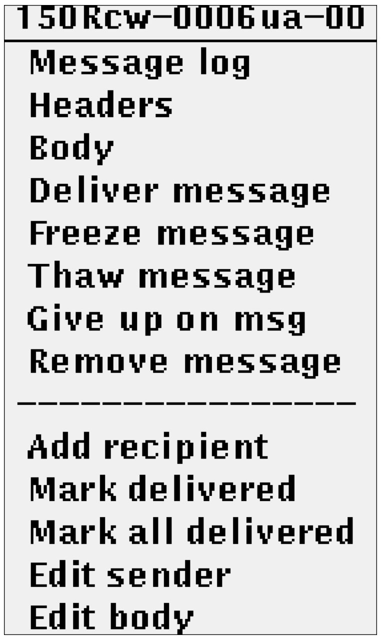


Figure 21-2. Monitor menu

The title of the menu is the message ID, and it contains entries that act as follows:

Message log
: The contents of the message log for the message are displayed in a new text window.

Headers
Information from the spool file containing the envelope information and headers is displayed in a new text window.

Body
The contents of the spool file containing the body of the message are displayed in a new text window. There is a default limit of 20KB to the amount of data displayed. This can be changed by setting `EXIMON_BODY_MAX`.

Deliver message
A call to Exim is made using the *-M* option to request delivery of the message. This call causes an automatic thaw if the message is frozen. The *-v* option is also set and the output from Exim is displayed in a new text window. The delivery is run in a separate process to avoid holding up the monitor while the delivery proceeds.

Freeze message
A call to Exim is made using the *-Mf* option to request that the message be frozen.

Thaw message
A call to Exim is made using the *-Mt* option to request that the message be thawed.

Give up on msg
A call to Exim is made using the *-Mg* option to request that Exim gives up trying to deliver the message. A delivery failure report is generated for any remaining undelivered addresses.

Remove message
A call to Exim is made using the *-Mrm* option to request that the message be deleted from the system without generating any failure reports.

Add recipient
A dialog box is displayed into which a recipient address can be typed. Pressing Enter causes a call to Exim to be made using the *-Mar* option (to request that an additional recipient be added to the message), unless the entry box is empty, in which case no action is taken.

Mark delivered
A dialog box is displayed into which a recipient address can be typed. Pressing Enter causes a call to Exim to be made using the *-Mmd* option (to mark the given recipient address as already delivered), unless the entry box is empty, in which case no action is taken.

Mark all delivered

A call to Exim is made using the *-Mmad* option to mark all recipient addresses as already delivered.

Edit sender

A dialog box is displayed initialized with the current sender's address for you to edit. Pressing Enter causes a call to Exim to be made using the *-Mes* option (to replace the sender address), unless the entry box is empty, in which case no action is taken.

Edit body

A new xterm process is forked in which a call to Exim is made using the *-Meb* option in order to allow the body of the message to be edited. Note that the first line of the body file is the name of the file, and this should never be changed.

In cases when a call to Exim is made, the actual command used is reflected in a new text window by default, but this command can be turned off for all except the delivery action by setting `EXIMON_ACTION_OUTPUT=no` in the environment. However, if the call results in output from Exim (in particular, if the command fails) a window containing the command and the output is displayed. Otherwise, the results of the action are normally apparent from the log and queue displays. The latter is automatically updated for actions such as freezing and thawing, unless `EXIMON_ACTION_QUEUE_UPDATE=no` has been set. In this case, the Update button has to be used to force an update to the display after freezing or thawing.

In any text window that is displayed as a result of a menu action, the normal cut-and-paste facility is available and searching can be carried out using CTRL-R and CTRL-S, as described earlier for the log tail window.

Maintaining Alias and Other Datafiles

Although Exim uses just one runtime configuration file, this generally refers to other files containing various kinds of data. If you update any of these files, Exim processes will pick up the new contents immediately; you do not need to take special action. However, if you update the runtime configuration itself, you need to send a `SIGHUP` signal to the daemon process. You need to be *root* or *exim* to do this, using a command such as:

```
kill -HUP `cat /var/spool/exim/exim-daemon.pid`
```

This command causes it to restart and reread the configuration. It is a good idea to check the log after you have done this to ensure that the daemon has restarted successfully. All Exim processes other than the daemon are short-lived, so as new ones start, they will see the new configuration.

Maintaining DBM Files

If you are using DBM files for aliases or any other data, you need to rebuild them from source files if you want to change the data. A utility program called *exim_dbmbuild* is provided for doing this. It reads an input file in the format of an alias file and writes a DBM database using the lowercased alias names as keys, and the remainder of the information as data. The lowercasing can be prevented by calling the program with the *-nolc* option.

A terminating zero is included as part of the key string. This is expected by the `dbm` lookup type. However, if the option *-nozero* is given, *exim_dbmbuild* creates files without terminating zeros in either the key strings or the data strings. The `dbmnz` lookup type can be used with such files.

The program requires two arguments: the name of the input file (which can be a single hyphen to indicate the standard input) and the name of the output database. It creates the database under a temporary name and then renames the file(s) if all went well. If the native Berkeley DB interface is in use (common in free versions of Unix), the two filenames must be different because in this mode, the Berkeley DB functions create a single output file using exactly the name given. For example:

```
exim_dbmbuild /etc/aliases /etc/aliases.db
```

reads the system alias file and creates a DBM version of it in */etc/aliases.db*.

In systems that use the `ndbm` routines (mostly proprietary versions of Unix), DBM databases consist of two files with suffixes *.dir* and *.pag*. In this environment, the suffixes are added to the second argument of *exim_dbmbuild*, so it can be the same as the first.

The program outputs a warning if it encounters a duplicate key. By default, only the first of a set of duplicates is used; this makes it compatible with `lsearch` lookups. An option called *-lastdup* causes it to use the last instead. There is also an option *-nowarn*, which stops it from listing duplicate keys to the standard error stream. If any duplicates are encountered, the return code is 1, unless *-noduperr* is used. For other errors for which it does not actually make a new file, the return code is 2.

Hints Database Maintenance

Exim uses DBM files to hold the data for its hints databases. Under normal circumstances you do not have to worry very much about these, except in one respect: they need "tidying" periodically. Out-of-date information accumulates in the files and if they are not tidied, their size keeps on increasing. This is usually quite gradual, so that weekly tidying is often sufficient, though some sites prefer to do it daily. A utility program called *exim_tidydb* is provided to do this job.

There is also a utility for dumping the contents of a hints database and one for making modifications. However, you are most unlikely to want to use either of them, so they are not described here. The reference manual has details if you need them.

The *exim_tidydb* utility program requires two arguments. The first specifies the name of Exim's spool directory, and the second is the name of the database that *exim_tidydb* is to operate on. These names are as follows:

`retry`*:*
: The database of retry information.

`reject`*:*
: The database of information about rejected messages.

`wait-`*`transport-name`*`:`
: Databases of information about messages waiting for remote hosts, using particular transports. Usually there is only one remote transport, so there is just one such database with a name such as `wait-remote_smtp`.

`serialize-`*`transport-name`*`:`
: Databases of information about current connections to hosts that are restricted to one connection at a time, using particular transports.

`serialize-etrn-runs`*:*
: Database of information about current queue runs started by the `ETRN` command when `smtp_etrn_serialize` is set.

If *exim_tidydb* is run with no options, it removes all records from the database that are more than 30 days old. For example:

```
exim_tidydb  /var/spool/exim  retry
```

The cutoff date can be altered with the *-t* option, which must be followed by a time. For example, to remove all records older than a week from the retry database, use:

```
exim_tidydb -t 7d /var/spool/exim retry
```

Some of the hints databases contain data pertaining to specific messages. For these, the *-f* option can also be used. This option causes a check to be made to ensure that message IDs in database records are those of messages still on the queue. The hints concerning messages that no longer exist are removed.

The *exim_tidydb* utility outputs comments on the standard output whenever it removes information from the database. It is suggested that it be run periodically on all databases, but at a quiet time of day, since it requires a database to be locked (and therefore be inaccessible to Exim) while it does its work. It can be run as the Exim user.

For example, if you are not using host or ETRN serialization and have just one remote transport called **remote_smtp**, these commands would be appropriate:

```
exim_tidydb -f /var/spool/exim retry            >/dev/null
exim_tidydb -f /var/spool/exim reject           >/dev/null
exim_tidydb -f /var/spool/exim wait-remote_smtp >/dev/null
```

You can put these commands in a file (called */usr/exim/bin/tidy_alldb*, for example) and have it run daily by a root *crontab* entry such as:

```
10 3 * * * su exim -c /usr/exim/bin/tidy_alldb
```

You have to make sure that *exim_tidydb* can be found, either by using an absolute pathname or by setting PATH in the script. Alternatively, you can use a *crontab* entry for *exim* instead of *root*, and, of course, you can put the individual commands directly into the *crontab* if you want to.

Mailbox Maintenance

Now and again there are occasions when you want to prevent any new messages from being delivered into a user's mailbox because you want to carry out maintenance activity on it, or investigate a problem. You could modify Exim's configuration file to defer deliveries to that user, but then you would have to remodify it afterwards, and in any case, this would not prevent user agents from modifying the mailbox. A better approach is to lock the mailbox.*

The *exim_lock* utility locks a mailbox file using the same algorithm as Exim. This prevents modification of a mailbox by Exim or a user agent. The utility requires the name of the file as its first argument. If the locking is successful, the second argument is run as a command. If there is no second argument, the value of the SHELL environment variable is used; if this is unset or empty, */bin/sh* is run. When the command finishes, the mailbox is unlocked and the utility ends. For example:

```
exim_lock /var/spool/mail/spqr
```

runs an interactive shell while the file is locked, whereas:

```
exim_lock -q /var/spool/mail/spqr <<End
some commands
End
```

* Locking only works, of course, if the mailbox is a single file. If you are using multifile mailboxes, another approach must be found.

runs a specific noninteractive sequence of commands while the file is locked. The *-q* ("quiet") option suppresses verification output. A single command can be run while the file is locked by a command such as:

```
exim_lock -q /var/spool/mail/spqr "cp /var/spool/mail/spqr /some/where"
```

Note that if a command is supplied, it must be entirely contained within the second argument; hence the quotes.

Without *-q*, some verification output is written. More detailed verification of the locking process can be requested with *-v*. There are also some options that are similar to the options on the **appendfile** transport used to control the way the mailbox is locked. Unless you have made changes to the locking options of **appendfile**, you should not need to specify any of these options.

-fcntl:
: Use `fcntl()` locking on the open mailbox.

-interval:
: Interval to sleep between retries, default 3 (seconds).

-lockfile:
: Create a lock file before opening the mailbox.

-mbx:
: Lock the mailbox using MBX rules.

-retries:
: Specifies the number of times to retry (default 10).

-timeout:
: Specifies the timeout value for `fcntl()` locking.

If none of *-fcntl*, *-lockfile* or *-mbx* are given, the default is to create a lock file and also use `fcntl()` locking on the mailbox, which is the same as Exim's default. The use of *-fcntl* requires that the file be writable; the use of *-lockfile* requires that the directory containing the file be writable. Locking by lock file does not last forever; Exim assumes that a lock file is expired if it is more than 30 minutes old. The *-mbx* option is mutually exclusive with *-fcntl*.

22

Building and Installing Exim

So far, we have talked only about how to use Exim, assuming that it is already installed on your system. We have not yet covered how to get it there. There are three possibilities:

- Some operating systems (for example, Debian GNU/Linux) are now distributed with Exim already installed. If yours is one of these, you do not need to do anything, unless you want to use some of the optional code that is not included in your binary or you want to upgrade to a later release. If you do, you will have to fetch the source and compile it yourself.
- Some operating systems (for example, FreeBSD) have a standardized "ports" mechanism, with a simple command that fetches the Exim source, compiles it with a particular set of options, and installs it for you. Again, if the options are suitable, you need do no more; if not, you must recompile, but in this scenario, the source is already available on your host. However, if you want to upgrade to a later release, you will have to fetch a new source.
- If neither of these apply to you, you will have to fetch a copy of the source yourself, and then compile and install it.

There is no general repository of binary distributions. One reason for this is there are a number of choices to be made when compiling Exim; for example, whether to include support for database lookups, and if so, for which database. Thus a single binary (even for one operating system) would suit only a few people.

This chapter describes the process of building and installing Exim from the source distribution.

Prerequisites

You need to have a working ANSI/ISO C compiler installed before you can build Exim. If you do not have a compiler, either consult your vendor or consider installing *gcc*, the GNU C compiler. You can find information about *gcc* at *http://www.fsf.org/order/ftp.html*. If you want to build the Exim monitor, the X Window system libraries and header files must also be available.

You will need *gunzip* (or *bunzip*) and *tar* to unpack the source. The building process assumes the availability of standard Unix tools such as *make* and *sed*. You do not need Perl in order to build or run Exim, but as some of the associated utilities are Perl scripts, it is a good idea to make sure that it is installed as well.

Finally, even if you do not use DBM lookups in your configuration, Exim requires a DBM library, because it uses DBM files for holding its hints databases. Licensed versions of Unix normally contain a library of DBM functions operating via the `ndbm` interface. Free versions of Unix vary in what they contain as standard. Some older versions of Linux have no default DBM library at all, and different distributors have chosen to include different libraries. However, the more recent releases of all the free operating systems seem to have standardized on the Berkeley DB library.

The original Berkeley DB package reached Version 1.85 before being superseded by Release 2 and then Release 3. The older versions are no longer maintained. You can find information about Berkeley DB at *http://www.sleepycat.com*.

Fetching and Unpacking the Source

The "Availability" link on the Exim home page, at *http://www.exim.org*, leads to a page containing a list of sites from which the source may be downloaded. The home page also contains information about the status of different versions. You can use your browser to download a distribution into an appropriate directory such as */usr/source/exim*. The distribution is a single compressed *tar* file, for example:

```
/usr/source/exim/exim-3.22.tar.gz
```

Move to that directory, and unpack Exim using *gunzip* and *tar*:*

```
$ cd /usr/source/exim
$ gunzip exim-3.22.tar.gz
$ tar -xf exim-3.22.tar
```

* The distribution is available in *bzip2* as well as *gzip* format; the former is substantially smaller, and therefore quicker to download. The final file extension is *.bz2*, instead of *.gz*, and you decompress it using *bunzip* instead of *gunzip*.

You should now have a directory called *exim-3.22*, which is the distribution file tree. You can delete the *tar* file to save space, and then move into the distribution directory:

```
$ rm exim-3.22.tar
$ cd exim-3.22
```

You should see the following files:

`CHANGES`	information about changes
`LICENCE`	the GNU General Public License
`Makefile`	top-level makeffile
`NOTICE`	conditions for the use of Exim
`README`	list of files, directories, and simple build instructions

Other files whose names begin with *README* may also be present. The following subdirectories should be present:

`OS`	OS-specific files
`doc`	documentation files
`exim_monitor`	source files for the Exim monitor
`scripts`	scripts used in the building process
`src`	source files for Exim and some utilities
`util`	independent utilities

The *doc* directory contains a copy of the reference manual as a plain text file called *spec.txt*. This is provided more for convenient searching than for sequential reading. You can download copies of the manual in various other formats; to save space, these are not included in the distribution. PostScript or PDF is best if you want to make a printed copy, whereas HTML and Texinfo are indexed formats for reading online.* The *doc* directory also contains information about changes and new additions to Exim. A complete list of runtime and build-time options can be found in the file *doc/OptionLists.txt*.

Some utilities are contained in the *src* directory and are built with the Exim binary; those distributed in the *util* directory are things such as the log file analyzer, which does not depend on any compile-time configuration.

Configuration for Building

The configuration that you set up for building Exim is contained in a directory called *Local*, so the first thing you have to do is to create that directory:

```
$ mkdir Local
```

* See the section "Installing Documentation in Info Format," later in this chapter for more on the Texinfo documentation.

The configuration for Exim itself must be created in a file called *Local/Makefile*, and the configuration for the Exim monitor in a file called *Local/eximon.conf*. You should never need to modify any of the original distribution files. If something in your operating system requires a different setting to the one in the file in the *OS* directory for your system, do not modify the default setting; instead insert an overriding setting in *Local/Makefile*. If you do things this way, you will be able to copy the contents of the *Local* directory and reuse them when the time comes to build the next release.

The Contents of Local/Makefile

The contents of *Local/Makefile* are a series of settings such as:

```
BIN_DIRECTORY=/usr/exim/bin
```

The settings can be in any order, and you can insert comment lines starting with # if you wish. A template for *Local/Makefile* is supplied in *src/EDITME*. It contains sample settings and comments describing what they are for. One way of creating your *Local/Makefile* is to copy the template and then edit the copy:

```
$ cp src/EDITME Local/Makefile
$ vi Local/Makefile
```

However, you can, of course, create *Local/Makefile* from scratch. There are a number of different types of settings that it can contain:

Mandatory
: These settings are necessary, or else Exim will not build.

Drivers
: These settings specify which drivers are included in the binary.

Modules
: These settings specify which optional modules (for example, lookup types) are included in the binary.

Recommended
: There are some values that can be set either at build time or in the runtime configuration. It is recommended they be set here if possible.

Optional
: These settings are simply a matter of choice.

System
: These settings depend on the configuration of your operating system.

The contents of *Local/Makefile* are combined with files from the *OS* directory to arrive at the settings to be used for building Exim in the following way:

- *OS/Makefile-Default* contains a common list of default settings. For example, it contains:

  ```
  CC=gcc
  ```

  to specify *gcc* as the default C compiler.
- Settings in the OS-specific *Makefile* can override the common defaults for a particular operating system. For example, in *OS/Makefile-IRIX65*, there is:

  ```
  CC=cc
  ```

  that specifies *cc* as the default C compiler when building under Irix 6.5.
- Finally, settings in *Local/Makefile* can override anything previously set. Suppose you wanted to use *gcc* under Irix 6.5 after all. The right way to do this is not to delete the setting in *OS/Makefile-IRIX65*, but instead to add:

  ```
  CC=gcc
  ```

  in *Local/Makefile*, so as to leave the distribution files unchanged.

Some option settings are for use in special cases, and are rarely needed. The following sections cover briefly those that are more commonly required. The *src/EDITME* and *OS/Makefile-Defaults* files contain more details, in the form of extended comments.

Mandatory Makefile Settings

There are only two settings that you must include in *Local/Makefile*. They specify the directory into which Exim will be installed, and the location of its runtime configuration file. For example:

```
BIN_DIRECTORY=/usr/exim/bin
CONFIGURE_FILE=/usr/exim/configure
```

However, if you build Exim with just these two settings, it will not be very useful, because the resulting binary contains no drivers and no code for any lookup types. It would be capable of receiving messages, but not delivering them.

Driver Choices in the Makefile

In practice, most people take the default settings in *src/EDITME* when it comes to choosing which directors, routers, and transports to include. These are as follows:

```
DIRECTOR_ALIASFILE=yes
DIRECTOR_FORWARDFILE=yes
DIRECTOR_LOCALUSER=yes
```

```
DIRECTOR_SMARTUSER=yes

ROUTER_DOMAINLIST=yes
ROUTER_LOOKUPHOST=yes

TRANSPORT_APPENDFILE=yes
TRANSPORT_AUTOREPLY=yes
TRANSPORT_PIPE=yes
TRANSPORT_SMTP=yes
```

If you want to include support for SMTP authentication, you must add one or both of the following:

```
AUTH_CRAM_MD5=yes
AUTH_PLAINTEXT=yes
```

Module Choices in the Makefile

The default settings for lookup types are:

```
LOOKUP_DBM=yes
LOOKUP_LSEARCH=yes
```

which cause the inclusion of code for `lsearch` and `dbm` lookup types. Other possibilities are as follows:

```
LOOKUP_CDB=yes
LOOKUP_DNSDB=yes
LOOKUP_LDAP=yes
LOOKUP_MYSQL=yes
LOOKUP_NIS=yes
LOOKUP_NISPLUS=yes
LOOKUP_PGSQL=yes
```

Apart from `LOOKUP_DNSDB`, you should set these only if you have the relevant software installed on your system. It is usually also necessary to specify where the appropriate library and include files may be found. For example, if you want to include support for MySQL, you might use:

```
LOOKUP_MYSQL=yes
LOOKUP_INCLUDE=-I /usr/local/mysql/include
LOOKUP_LIBS=-L/usr/local/lib -lmysqlclient
```

Recommended Makefile Settings

A few important settings can be specified either at build time or at runtime. It is recommended that these are set at build time if their values are fixed, for two reasons:

- Runtime settings can be lost accidentally, which might lead to serious misbehavior.

- A change to the log file path at runtime cannot take effect until the runtime configuration has been read. If there is a serious problem before this (for example, an inability to read the runtime configuration), Exim cannot log it to the correct place, and maybe cannot log it at all.

If it is so dangerous, why can these settings be changed at runtime? There are two reasons:

- Some administrators need to distribute copies of the binary to a number of machines with slightly different requirements. They are prepared to accept the risk.
- It makes certain kinds of testing easier.

Examples of the five settings that are in this recommended category are as follows:

```
EXIM_UID=42
EXIM_GID=42
LOG_FILE_PATH=/var/log/exim_%slog
SPOOL_DIRECTORY=/var/spool/exim
SPOOL_MODE=0640
```

`EXIM_UID` and `EXIM_GID` define the uid and gid for the Exim user; that is, the identity under which it runs when it does not need *root* privilege.

You do not need to set `LOG_FILE_PATH` at all if you are happy with the default, which is to use a subdirectory of the spool directory, equivalent in this example to:

```
LOG_FILE_PATH=/var/spool/exim/log/%slog
```

You do not need to set `SPOOL_MODE` if you are happy with the default value of 0600. Setting it to 0640 allows members of the Exim group to read spool files, which is necessary for running the Exim monitor.

A Plausible Minimal Makefile

Here is an example of a minimal *Local/Makefile* that includes the recommended settings as well as the default drivers and lookups:

```
BIN_DIRECTORY=/usr/exim/bin
CONFIGURE_FILE=/usr/exim/configure

DIRECTOR_ALIASFILE=yes
DIRECTOR_FORWARDFILE=yes
DIRECTOR_LOCALUSER=yes
DIRECTOR_SMARTUSER=yes

EXIM_GID=42
EXIM_UID=42
```

```
LOOKUP_DBM=yes
LOOKUP_LSEARCH=yes

ROUTER_DOMAINLIST=yes
ROUTER_LOOKUPHOST=yes

SPOOL_DIRECTORY=/var/spool/exim
SPOOL_MODE=0640

TRANSPORT_APPENDFILE=yes
TRANSPORT_AUTOREPLY=yes
TRANSPORT_PIPE=yes
TRANSPORT_SMTP=yes
```

Building Exim with this file produces a usable binary that can do straightforward mail delivery.

System-Related Makefile Settings

The C compiler is called either *cc* or *gcc*, or sometimes something else again, and different compilers accept different option settings. The system-related settings allow you to specify the name of your compiler and its options, for example:

```
CC=cc
CFLAGS=-Otax -4
```

On some operating systems, additional libraries must be specified. For example, Solaris keeps the socket-related functions in a separate library. The OS-dependent makefiles use `LIBS` for these settings. For example, the Solaris file contains:

```
LIBS=-lsocket -lnsl -lkstat
```

If you want to add yet more libraries of your own, you should use `EXTRALIBS` rather than `LIBS`, but you can of course use `LIBS` if you want to override what is in the distribution files.

The settings in `LIBS` and `EXTRALIBS` are used for every binary that is built, which includes some of the utilities. If you want to restrict certain libraries to just the Exim binary or just the *eximon* binary, you can use `EXTRALIBS_EXIM` and `EXTRALIBS_EXIMON`, respectively.

Settings for the DBM library are also commonly required:

```
USE_DB=yes
DBMLIB=-ldb
```

The defaults for these settings are taken from the system-specific makefiles in the *OS* directory, so, in most cases, you should not need to set them in *Local/Makefile*.

If you are not going to use `timestamps_utc` in the runtime configuration (in other words, if you want Exim to use wallclock time for its timestamps), you might want to set `TIMEZONE_DEFAULT` in *Local/Makefile*, for example:

```
TIMEZONE_DEFAULT=EST
```

This will provide the default value for the `timezone` option (see the section "Timestamps," in Chapter 19, *Miscellany*). If it is not included, the value of the `TZ` environment variable at the time Exim is built is used.

Optional Settings in the Makefile

The remaining settings in *Local/Makefile* are simply a matter of choice. For example:

```
EXICYCLOG_MAX=28
```

specifies that the *exicyclog* utility should keep a maximum of 28 old log files (the default is 10). The comments in *src/EDITME* explain what the settings do in each case.

Configuration for Building the Exim Monitor

The Exim monitor is built along with Exim, and if you want to do this, you must set up a suitable configuration for it. There are only two things that are mandatory:

- In the main configuration file, *Local/Makefile*, you must include:

  ```
  EXIM_MONITOR=eximon.bin
  ```

  If this setting is not present, the monitor is omitted from the building process. You may also need to specify the whereabouts of the X11 library and include files if the defaults are not correct. For example:

  ```
  X11=/opt/X11R6.3
  XINCLUDE=-I$(X11)/include
  XLFLAGS=-L$(X11)/lib
  X11_LD_LIB=$(X11)/lib
  ```

- You must create *Local/eximon.conf*, which is a configuration file containing your choices.

A commented template for *Local/eximon.conf* is supplied in *exim_monitor/EDITME*. In this case, there are no mandatory settings, so the file can be completely empty, though it must exist. You can set up an empty file with the command:

```
touch Local/eximon.conf
```

The available settings allow you to change the size of the window and the appearance of some of the data. Descriptions of each setting appear as comments in *exim_monitor/EDITME*.

Building Exim for Multiple Systems

If you are building Exim for just a single operating system on a single host, you can skip this section entirely. If, on the other hand, you have a single source directory that is accessible to a number of hosts that are running different operating systems, or using different hardware architectures, you may want to set different values for different cases.

So far, we have talked about a single *Local/Makefile* containing all the local settings. You can, in fact, supply separate files for each operating system, each hardware architecture, and each combination of operating system and architecture, if you so wish. These are optional files that are consulted in addition to *Local/Makefile* only if they exist. The full list of all possible files is as follows:

```
Local/Makefile
Local/Makefile-ostype
Local/Makefile-archtype
Local/Makefile-ostype-archtype
```

where *ostype* is the operating system type (for example, `Linux`), and *archtype* is the hardware architecture type (for example, `i386`). The files are used in that order. In other words, settings in *Local/Makefile* apply to all cases, but can be overridden by settings in *Local/Makefile-ostype*, which in turn can be overriden by the other two files. Thus, a single set of files can contain the correct settings for all the different cases, with a minimum of repetition.

A similar scheme is used for the Exim monitor, where the filenames are as shown:

```
Local/eximon.conf
Local/eximon.conf-ostype
Local/eximon.conf-archtype
Local/eximon.conf-ostype-archtype
```

The values that are used for *ostype* and *archtype* are obtained from scripts called *scripts/os-type* and *scripts/arch-type*, respectively. If either of the environment variables `EXIM_OSTYPE` or `EXIM_ARCHTYPE` is set, their values are used instead, thereby providing a means of forcing particular settings. Otherwise, the scripts try to find suitable values by running the *uname* command. If this fails, the shell variables `OSTYPE` and `ARCHTYPE` are inspected. A number of ad hoc transformations are then applied, to produce the standard names that Exim expects. You can run these scripts directly from the shell in order to find out what values will be used on your system.

The toplevel makefile copes with rebuilding Exim correctly if any of the configuration files are edited. However, if an optional configuration file is deleted, it is necessary to *touch* the associated nonoptional file (that is, *Local/Makefile* or */Local/eximon.conf*) before rebuilding.

The Building Process

Once you have created the appropriate configuration files in the *Local* directory, you can run the building process by the single command:

```
$ make
```

The first thing this does is to create a "build directory" whose name is *build-ostype-archtype*; for example, *build-SunOS5-5.8-sparc*. Links to the source files are installed in this directory, and all the files that are created while building are written here. This way of doing things means that you can build Exim for different operating systems and different architectures from the same set of shared source files if you want to.

If this is the first time *make* has been run, it calls a script that builds a makefile inside the build directory, using the configuration files from the *Local* directory.* The new makefile is passed to another instance of *make*, which does the real work. First, it builds a C-header file called *config.h*, using values from *Local/Makefile*, and then it builds a number of utility scripts. Next, it compiles and links the binaries for the Exim monitor (if configured), a number of utilities, and finally Exim itself. If all goes well, the last line of output on your screen should be:

```
>>> exim binary built
```

If you have problems building Exim, check for any comments there may be in the *README* file concerning your operating system, and also take a look at the FAQ, where some common problems are covered. It is available from any of the FTP sites, and also via *http://www.exim.org*.

Installing Exim

The command:

```
$ make install
```

runs a script called *scripts/exim_install*, which copies the binaries and scripts into the directory whose name is specified by `BIN_DIRECTORY` in *Local/Makefile*. Files are copied only if they are newer than any versions already in the directory, and when they are copied, old versions are renamed by adding *.O* to their names.

* The command *make makefile* can be used to force a rebuild of the makefile in the build directory, should this ever be necessary. If you make changes to *Local/Makefile*, it is automatically rebuilt when next you run *make*.

You need to be *root* when you run this command, because, for most configurations, the main Exim binary is required to be owned by *root* and have the setuid bit set. The install script therefore tries to set *root* as the owner of the main binary and to make it setuid. If you want to see what the script will do before running it for real, run it from the build directory, using the *-n* option (for which *root* privilege is not needed):

```
    $ (cd build-SunOS5-5.5.1-sparc; ../scripts/exim_install -n)
```

The *-n* option causes the script to output a list of the commands it would obey, without actually obeying any of them.

If the runtime configuration file, as defined by `CONFIGURE_FILE` in *Local/Makefile*, does not exist, the default configuration file *src/configure.default* is copied there by the installation script. If a runtime configuration file already exists, it is left alone.

The default configuration uses the local host's name as the only local domain, and is set up to do local deliveries into the shared directory */var/mail*, running as the local user. Aliases in */etc/aliases* and *.forward* files in users' home directories are supported. Remote domains are routed using the DNS, with delivery over SMTP.

You do not need to create the spool directory when installing Exim. When it starts up, it creates the spool directory if it does not exist. If a specific Exim uid and gid are specified, these are used for the owner and group of the spool directory. Subdirectories are automatically created in the spool directory as necessary.

If you are installing Exim on a system that is running some other MTA, installing the files by running *make install* does not of itself cause Exim to supersede the other MTA. Once you get this far, it is all ready to go, but it still needs to be "turned on" before it will start handling your mail. Before you take this final step, it is a good idea to do some testing.

Testing Before Turning On

When all the files are in place, you can run various tests, including passing messages to Exim directly and having it deliver them. You can then inspect the log files or run the monitor if you wish. The one thing you cannot do while another MTA is running is to run a daemon on the standard SMTP port, but if you wish to test the daemon, an alternative port can be used.

First, check that the runtime configuration file is syntactically valid by running the command:

```
    exim -bV
```

If there are any errors in the configuration file, Exim outputs error messages, which are also written to the panic log. Otherwise it just outputs the version number and build date. Directing and routing tests can be done by using the address testing option. For example:

```
exim -v -bt user@your.domain
```

checks that it recognizes a local mailbox, and:

```
exim -v -bt user@somewhere.else.example
```

a remote one. Then try getting it to deliver mail, both locally and remotely. This can be done by passing messages directly to Exim, without going through a user agent. For example:

```
exim postmaster@your.domain
From: user@your.domain
To: postmaster@your.domain
Subject: Testing Exim

This is a test message.
.
```

If you encounter problems, look at Exim's log files to see if there is any relevant information there, and use the *-bp* option to see if the message is still on Exim's queue. Another source of information is running Exim with debugging turned on. With *-d2*, for example, the sequence of directors or routers that process an address is output. If a message is stuck on Exim's spool, you can force a delivery with debugging turned on by a command of the form:

```
exim -d2 -M 13A918-0000iT-00
```

One specific problem that has shown up on some sites is the inability to do local deliveries into a single shared mailbox directory that does not have the "sticky bit" set on it.* By default, Exim tries to create a lock file before writing to a mailbox file, and if it cannot create the lock file, the delivery is deferred. You can get round this either by setting the "sticky bit" on the directory, or by setting a specific group for local deliveries and allowing that group to create files in the directory (see the comments above the **local_delivery** transport in the default configuration file). For further discussion of locking issues, see the section "Locking a File for Appending" in Chapter 9.

You can test a daemon by running it on a nonstandard port by a command such as the following:

```
exim -bd -oX 1225
```

* An explanation of the "sticky bit" is given in the section "Mailbox location," in the description of the **appendfile** transport in Chapter 9, *The Transports*.

and using *telnet* to connect to port 1225.* However, if you want to test out policy controls for incoming mail, the *-bh* option is better, because it allows you to simulate an incoming call from any IP address you like.

Testing a new version on a system that is already running Exim can most easily be done by building a binary with a different `CONFIGURE_FILE` setting. From within the runtime configuration, all other file and directory names that Exim uses can be altered, in order to keep it entirely clear of the production version.

Turning Exim On

The conventional pathname that is used to call the MTA on a Unix system is either */usr/sbin/sendmail* or */usr/lib/sendmail*. In some cases, both paths exist, usually pointing to the same file. User agents use one or the other of these names to send messages, and there is usually a reference from one of the system boot scripts that starts a listening daemon.

The process of "turning Exim on" consists of changing these paths so that they refer to Exim instead of to the previous MTA. This is normally done by renaming the existing file and setting up a symbolic link. You need to be *root* to do this. It is also a good idea to remove the setuid bit from the previous MTA, and/or make it inaccessible. For example:

```
$ mv /usr/sbin/sendmail /usr/sbin/sendmail.old
$ chmod 0600 /usr/sbin/sendmail.old
$ ln -s /usr/exim/bin/exim /usr/sbin/sendmail
```

Once this is done, any program that calls */usr/sbin/sendmail* actually calls Exim. Your Exim installation is now "live." Check it by sending a message from your favourite user agent.

There are two more things to do once you have Exim running on your host: set up *cron* jobs to cycle the log files (unless you are using *syslog* only) and tidy the hints databases from time to time, as described in the section "Cycling Log Files," and the section "Hints Database Maintenance," in Chapter 21, *Administering Exim*.

* It is often useful to add *-d* to such a command; it outputs minimal debugging, and, in addition, it leaves the daemon connected to the terminal, so you can easily kill it with CTRL-C.

Installing Documentation in Info Format

Some operating systems have standardized on the GNU *info* system for documentation, and if yours is one of these, you probably want to install Exim's documentation in this format. You can arrange for this to happen as part of the Exim installation process by making a few preliminary preparations.

The source of the *info* version of the documentation is not included in the Exim distribution, because not everybody wants it, so you have to fetch it separately. The site from which you obtained Exim should also have a file with a name such as this:

```
exim-texinfo-3.20.tar.gz
```

This unpacks into three files called:

```
exim-texinfo-3.20/doc/filter.texinfo
exim-texinfo-3.20/doc/oview.texinfo
exim-texinfo-3.20/doc/spec.texinfo
```

The version number will always end in a zero, because the main documentation is not updated for intermediate releases where the version number ends with a nonzero digit. Copy or move these into the *doc* directory of the source tree that you are using:

```
$ mv exim-texinfo-3.20/doc/* exim-3.22/doc
```

Then add to your *Local/Makefile* a line of the form:

```
INFO_DIRECTORY=/usr/local/info
```

to define the location of the *info* files on your system. Once this is done, running the following:

```
$ make install
```

automatically builds the *info* files from the *texinfo* sources, and installs them in */usr/local/info*.

Upgrading to a New Release

Once you have fetched and unpacked the source of a new release, you should read the file called *README.UPDATING*. This contains information about changes that might affect the way Exim runs or that require changes to the configuration. Most releases of Exim are entirely backwards-compatible with their predecessors, though there was an incompatible change to the runtime configuration between Release 2.12 and Release 3.00, and a further big change is planned in due course for Exim 4.

You can normally just copy the files in your *Local* directory to the source tree for the new release in order to build it with the same settings as before. You may, of course, need to add to them in order to take advantage of new features that require configuration at build time.

After you have built a new release, provided that it is compatible with the old run-time configuration, you can install it "on the fly" without having to stop anything. There has only been one new release where this was not possible. If it happens again, you can be sure that *README.UPDATING* will warn you about it, and tell you how to proceed. Otherwise, just run:

```
$ make install
```

Once this has been done, programs that call the MTA will immediately start using the new version instead of the old. However, the daemon process will continue to run the old version until you tell it to reload itself by sending it a HUP signal.

A
Summary of String Expansion

This appendix contains a list of all the available expansion items, conditions, and variables, in alphabetical order in each case, with brief descriptions. A more detailed discussion of the expansion items and conditions can be found in Chapter 17, *String Expansion.*

Expansion Items

The following items are recognized in expanded strings. Whitespace may be used between subitems that are keywords or substrings enclosed in braces inside an outer set of braces, to improve readability.

`$`*`variable-name`* *or* `${`*`variable-name`*`}`
: The contents of the named variable are substituted. An unknown variable name causes an error.

`${domain:`*`string`*`}`
: The string is expanded; it is then interpreted as an RFC 822 address and the domain is extracted from it.

`${escape:`*`string`*`}`
: If the expanded string contains any nonprinting characters, they are converted to escape sequences starting with a backslash.

`${expand:`*`string`*`}`
: The string is expanded twice.

`${extract{`*`key`*`} {`*`string`*`}}`
: The subfield identified by the key is extracted from the expanded string,

`${extract{`*`number`*`} {`*`separators`*`} {`*`string`*`}}`
: The subfield numbered *`number`* is extracted from the expanded string.

`${hash_`*`n_m`*`:`*`string`*`}`
: A textual hash of length *`n`* is generated, using characters from the first *`m`* characters of the concatenation of lowercase letters, uppercase letters, and digits. See also `nhash`.

`$header_`*`header-name`*`:` *or* `$h_`*`header-name`*`:`
: The contents of the named message header are substituted. If there is no such header, no error occurs, and nothing is substituted.

`${if `*`condition`*` {`*`string1`*`}{`*`string2`*`}}`
: If *`condition`* is true, *`string1`* is expanded; otherwise *`string2`* is expanded.

`${lc:`*`string`*`}`
: The letters in the expanded string are forced into lowercase.

`${length_`*`number`*`:`*`string`*`}`
: The initial *`number`* characters of the expanded string are substituted.

`${local_part:`*`string`*`}`
: The expanded string is interpreted as an RFC 822 address, and the local part is extracted from it.

`${lookup{`*`key`*`} `*`single-key-lookup-type`*` {`*`file`*`}{`*`string1`*`}{`*`string2`*`}}`
: The key is looked up in the given file using the given lookup type. If it is found, *`string1`* is expanded with $value containing the data; otherwise *`string2`* is expanded.

`${lookup `*`query-style-lookup-type`*` {`*`query`*`}{`*`string1`*`}{`*`string2`*`}}`
: The query is passed to the given query-style lookup. If it succeeds, *`string1`* is expanded with $value containing the data; otherwise *`string2`* is expanded.

`${mask:`*`IP address/bitcount`*`}`
: An IP address where all but the most significant *`bitcount`* bits are forced to zero is substituted, followed by */`bitcount`*.

`${nhash_`*`n`*`:`*`string`*`}`
: The string is expanded and then processed by a hash function that returns a numeric value in the range 0 to *`n-1`*.

`${nhash_`*`n_m`*`:`*`string`*`}`
: The string is expanded and then processed by a div/mod hash function that returns two numbers, separated by a slash, in the ranges 0 to *`n-1`* and 0 to *`m-1`*, respectively.

`${perl{`*`subroutine`*`}{`*`arg`*`}{`*`arg`*`} . . . }`
: The Perl subroutine is called with the given arguments, up to a maximum of eight. The arguments are first expanded.

`${quote:`*`string`*`}`
: The string is expanded and then substituted, in double quotes if it contains anything other than letters, digits, underscores, dots, and hyphens. Any occurrences of double quotes and backslashes are escaped with a backslash.

`${quote_`*`lookup-type`*`:`*`string`*`}`
: Lookup-specific quoting rules are applied to the expanded string.

`${rxquote:`*`string`*`}`
: A backslash is inserted before any nonalphanumeric characters in the expanded string.

`${sg{`*`subject`*`}{`*`regex`*`}{`*`replacement`*`}}`
: The regular expression is repeatedly matched against the expanded subject string, and for each match, the expanded replacement is substituted. $1, $2, and so on can be used in the replacement to insert captured substrings.

`${substr_`*`offset_length`*`:`*`string`*`}`
: A substring of length *`length`* starting at offset *`offset`* is extracted from the expanded string. Negative offsets count backwards from the end of the string.

`${tr{`*`subject`*`}{`*`string1`*`}{`*`string2`*`}}`
: The expanded *`subject`* is translated by replacing characters found in *`string1`* by the corresponding characters in *`string2`*.

`${uc:`*`string`*`}`
: The letters in the expanded string are forced into uppercase.

Expansion Conditions

The following conditions are available for testing by the `${if` item while expanding strings:

`!`*`condition`*
: Preceding any condition with an exclamation mark negates the result of the condition.

`symbolic operator` `{`*`string1`*`}{`*`string2`*`}`
: There are a number of symbolic operators for doing numeric comparisons. They are:

= equal to
== equal to
> greater than
>= greater than or equal to

< less than
<= less than or equal to

The two strings must take the form of optionally signed decimal integers, optionally followed by one of the letters K or M (in either upper- or lowercase), signifying multiplication by 1024 or 1024×1024, respectively.

`crypteq {`*`string1`*`}{`*`string2`*`}`

The `crypteq` condition has two arguments. The first is encrypted and compared against the second, which is already encrypted. This condition is included in the Exim binary if it is built to support any authentication mechanisms. Otherwise, it is necessary to define `SUPPORT_CRYPTEQ` in *Local/Makefile* to have `crypteq` included in the binary.

`def:`*`variable-name`*

This form of the `def` condition must be followed by the name of one of the expansion variables defined in the section "Expansion Variables," later in this appendix. The condition is true if the named expansion variable does not contain the empty string. The variable name is given without a leading dollar character. If the variable does not exist, the expansion fails.

`def:header_`*`header-name`*`:` *or* `def:h_`*`header-name`*`:`

This form of the `def` condition is true if a message is being processed and the named header exists in the message. No dollar appears before `header_` or `h_` in the condition, and the header name must be terminated by a colon if whitespace does not follow.

`eq {`*`string1`*`}{`*`string2`*`}`

The two substrings are first expanded. The condition is true if the two resulting strings are identical, including the case of letters.

`exists {`*`filename`*`}`

The substring is first expanded and then interpreted as an absolute path. The condition is true if the named file (or directory) exists.

`first_delivery`

This condition, which has no data, is true during a message's first delivery attempt. It is false during any subsequent delivery attempts.

`match {`*`string1`*`}{`*`string2`*`}`

The two substrings are first expanded. The second is treated as a regular expression and applied to the first. Because of the preexpansion, if the regular expression contains dollar or backslash characters, they must be escaped with backslashes. Care must also be taken if the regular expression contains braces (curly brackets). A closing brace must be escaped so that it is not taken as a premature termination of *`string2`*. It does no harm to escape opening braces, but this is not strictly necessary.

`pam {`*`string1`*`:`*`string2`*`:...}`

The *Pluggable Authentication Module* (PAM) module is initialized with the service name "exim" and the username taken from the first item in the colon-separated data string (that is, *`string1`*). The remaining items in the data string are passed over in response to requests from the authentication function. In the simple case, there will only be one request for a password, so the data will consist of two strings only.*

`queue_running`

This condition, which has no data, is true during delivery attempts that are initiated by queue runner processes, and false otherwise.

Combining Conditions

Two or more conditions can be combined using `and` and `or`:

`and {{`*`cond1`*`}{`*`cond2`*`}...}`

The subconditions are evaluated from left to right. The condition is true if all of the subconditions are true. If there are several `match` subconditions, the values of the numeric variables afterwards are taken from the last one. When a false subcondition is found, the following ones are parsed but not evaluated.

`or {{`*`cond1`*`}{`*`cond2`*`}...}`

The subconditions are evaluated from left to right. The condition is true if any one of the subconditions is true. When a true subcondition is found, the following ones are parsed but not evaluated. If there are several `match` subconditions, the values of the numeric variables afterwards are taken from the first one that succeeds.

Note that `and` and `or` are complete conditions on their own, and precede their lists of subconditions. Each subcondition must be enclosed in braces within the overall braces that contain the list. No repetition of `if` is used.

* Pluggable Authentication Modules (*http://ftp.kernel.org/pub/linux/libs/pam/*) are a facility that is available in the latest releases of Solaris and in some GNU/Linux distributions.

Expansion Variables

The variable substitutions available for use in expansion strings are as follows:

$0, $1, and so on
When a `matches` expansion condition succeeds, these variables contain the captured substrings identified by the regular expression during subsequent processing of the success string of the containing `if` expansion item. They may also be set externally by some other matching process that precedes the expansion of the string.

$address_file
When a message is directed to a specific file as a result of aliasing or forwarding, this variable holds the name of the file when the transport is running. For example, using the default configuration, if user `r2d2` has a *.forward* file containing:

```
/home/r2d2/savemail
```

then when the **`address_file`** transport is running, `$address_file` contains */home/r2d2/savemail*. At other times, the variable is empty.

$address_pipe
When a message is directed to a pipe, as a result of aliasing or forwarding, this variable holds the pipe command when the transport is running.

$authenticated_id
When a server successfully authenticates a client, it may be configured to preserve some of the authentication information in the variable $authenticated_id. For example, a user/password authenticator configuration might preserve the username for use in the directors or routers.

$authenticated_sender
When a client host has authenticated itself, Exim pays attention to the `AUTH=` parameter on the SMTP `MAIL` command. Otherwise, it accepts the syntax, but ignores the data. Unless the data is the string <>, it is set as the authenticated sender of the message, and the value is available during delivery in the $authenticated_sender variable.

$body_linecount
This variable holds the number of lines in the body of a message.

$caller_gid
This variable holds the group ID under which the process that called Exim was running. This is not the same as the group ID of the originator of a message (see $originator_gid). If Exim re-execs itself, this variable in the new incarnation normally contains the Exim gid.

$caller_uid

This variable holds the user ID under which the process that called Exim was running. This is not the same as the user ID of the originator of a message (see $originator_uid). If Exim re-execs itself, this variable in the new incarnation normally contains the Exim uid.

$compile_date

This variable holds the date on which the Exim binary was compiled.

$compile_number

The building process for Exim keeps count of the number of times it has been compiled. This serves to distinguish different compilations of the same version of the program.

$domain

When an address is being directed, routed, or delivered on its own, this variable contains the domain. In particular, it is set during user filtering, but not during system filtering, since a message may have many recipients and the system filter is called just once.

For remote addresses, the domain that is being routed can change as routing proceeds, as a result of router actions (see, for example, the `domainlist` router). However, the value of $domain remains as the original domain. The current routing domain can often be accessed by other means.

When a remote or local delivery is taking place, if all the addresses that are being handled simultaneously contain the same domain, it is placed in $domain. Otherwise, this variable is empty during delivery. Transports should be restricted to handling only one domain at once if its value is required at transport time. This is the default for local transports.

At the end of a delivery, if all deferred addresses have the same domain, it is set in $domain during the expansion of `delay_warning_condition`.

When an address rewriting configuration item is being processed, $domain contains the domain portion of the address that is being rewritten; it can be used in the expansion of the replacement address.

When the `smtp_etrn_command` option is being expanded, $domain contains the complete argument of the `ETRN` command.

$domain_data

When a director or a router has a setting of the `domains` option, and the domain is matched by a file lookup, the data obtained from the lookup is available during the running of the director or router, and any subsequent transport, as $domain_data.

$errmsg_recipient

This is set to the recipient address of an error message while Exim is creating it. It is useful if a customized error message text file is in use.

$home

A home directory may be set during a local delivery, either by the transport or by the director that handled the address. When this is the case, $home contains its value and may be used in any expanded options for the transport. The **forwardfile** director also makes use of $home. Full details are given in the section "The forwardfile Director," in Chapter 7, *The Directors*. When interpreting a user's filter file, Exim is normally configured so that $home contains the user's home directory. When running a filter test via the *-bf* option, $home is set to the value of the environment variable `HOME`.

$host

When a local transport is run as a result of routing a remote address, this variable is available to access the hostname that the router defined. A router may set up many hosts; in this case $host refers to the first one.

When used in a transport filter (see the section "Transport Filters," in Chapter 9, *The Transports*) $host refers to the host involved in the current connection.

When used in the client part of an authenticator configuration, or when the **smtp** transport is expanding its options for TLS authentication, $host contains the name of the server to which Exim is connected.

$host_address

This variable is set to the remote host's IP address whenever $host is set for a remote connection.

$host_lookup_failed

This variable contains "1" if the message came from a remote host and there was an attempt to look up the host's name from its IP address, but the attempt failed. Otherwise the value of the variable is "0."

$interface_address

For a message received over a TCP/IP connection, this variable contains the address of the IP interface that was used. See also the *-oMi* command-line option.

$key

When a domain, host, or address list is being searched, this variable contains the value of the key, so that it can be inserted into strings for query-style lookups. See Chapter 16, *File and Database Lookups*, for details.

$local_part

When an address is being directed, routed, or delivered on its own, this variable contains the local part. If a local part prefix or suffix has been recognized, it is not included in the value. When a number of addresses are being delivered in a batch by a local or remote transport, $local_part is not set.

When a message is being delivered to a **pipe**, **file**, or **autoreply** transport as a result of aliasing or forwarding, $local_part is set to the local part of the parent address.

When a configuration rewrite item is being processed, $local_part contains the local part of the address that is being rewritten.

$local_part_data

When a director or a router has a setting of the `local_parts` option and the local part is matched by a file lookup, the data obtained from the lookup is available during the running of the director or router, and any subsequent transport, as $local_part_data.

$local_part_prefix

When an address is being directed or delivered locally, and a specific prefix for the local part is recognized, it is available in this variable.

$local_part_suffix

When an address is being directed or delivered locally, and a specific suffix for the local part is recognized, it is available in this variable.

$localhost_number

This contains the expanded value of the `localhost_number` option. The expansion happens after the main options have been read.

$message_age

This variable is set at the start of a delivery attempt to contain the number of seconds since the message was received. It does not change during a single delivery attempt.

$message_body

This variable contains the initial portion of a message's body while it is being delivered, and is intended mainly for use in filter files. The maximum number of characters of the body that are used is set by the `message_body_visible` configuration option; the default is 500. Newlines are converted into spaces to make it easier to search for phrases that might be split over a line break.

$message_body_end

This variable contains the final portion of a message's body while it is being delivered. The format and maximum size are as for $message_body.

$message_body_size

When a message is being received or delivered, this variable contains the size of the body in bytes. The count starts from the character after the blank line that separates the body from the header lines. Newlines are included in the count. See also $message_size.

$message_headers

This variable contains a concatenation of all the header lines when a message is being processed. They are separated by newline characters.

$message_id

When a message is being received or delivered, this variable contains the unique message ID that is used by Exim to identify the message.

$message_precedence

When a message is being delivered, the value of any *Precedence:* header is made available in this variable. If there is no such header, the value is the null string.

$message_size

When a message is being received or delivered, this variable contains its size in bytes. The size includes those headers that were received with the message, but not those (such as *Envelope-to:*) that are added to individual deliveries. See also $message_body_size.

$n0 to $n9

These variables are counters that can be incremented by means of the *add* command in filter files.

$original_domain

When a top-level address is being processed for delivery, this contains the same value as $domain. However, if a "child" address (for example, generated by an alias, forward, or filter file) is being processed, this variable contains the domain of the original address. This differs from $parent_domain when there is more than one level of aliasing or forwarding. When more than one address is being delivered in a batch by a local or remote transport, $original_domain is not set.

Address rewriting happens as a message is received. Once it has happened, the previous form of the address is no longer accessible. It is the rewritten top-level address whose domain appears in this variable.

$original_local_part

This is the counterpart of $original_domain, and contains the local part of the original top-level address.

$originator_gid

This is the value of $caller_gid that was set when the message was received. For messages received via the command line, this is the gid of the sending user. For messages received by SMTP over TCP/IP, this is normally the gid of the Exim user.

$originator_uid

The value of $caller_uid that was set when the message was received. For messages received via the command line, this is the uid of the sending user. For messages received by SMTP over TCP/IP, this is normally the uid of the Exim user.

$parent_domain

This variable is empty, except when a "child" address (generated by aliasing or forwarding, for example) is being processed, in which case it contains the domain of the immediately preceding parent address.

$parent_local_part

This variable is empty, except when a "child" address (generated by aliasing or forwarding, for example) is being processed, in which case it contains the local part of the immediately preceding parent address.

$pipe_addresses

This is not an expansion variable, but is mentioned here because the string `$pipe_addresses` is handled specially in the command specification for the **pipe** transport and in transport filters. It cannot be used in general expansion strings, and provokes an "unknown variable" error if encountered.

$primary_hostname

This variable holds the value set in the configuration file, or the value determined by running the `uname()` function.

$prohibition_reason

This variable is set only during the expansion of prohibition messages. See the section "Customizing Prohibition Messages," in Chapter 13, *Message Reception and Policy Controls*, for details.

$qualify_domain

This variable holds the value set for this option in the configuration file.

$qualify_recipient

This variable holds the value set for this option in the configuration file, or if not set, the value of $qualify_domain.

$rbl_domain

This variable holds the name of the RBL domain that caused rejection during the expansion of the contents of the `prohibition_reason` option.

$rbl_text
This variable holds the contents of an RBL TXT record during the expansion of the contents of the `prohibition_reason` option.

$received_for
If there is only a single recipient address in an incoming message when the *Received:* header line is being built, this variable contains that address.

$received_protocol
When a message is being processed, this variable contains the name of the protocol by which it was received.

$recipients
This variable contains a list of envelope recipients for a message, but is recognized only in the system filter file, to prevent exposure of *BCC:* recipients to ordinary users. A comma and a space separate the addresses in the replacement text.

$recipients_count
When a message is being processed, this variable contains the number of envelope recipients that came with the message. Duplicates are not excluded from the count.

$reply_address
When a message is being processed, this variable contains the contents of the *Reply-To:* header if one exists, or otherwise the contents of the *From:* header.

$return_path
When a message is being delivered, this variable contains the return path; that is, the sender field that will be sent as part of the envelope. It is not enclosed in angle brackets. In many cases, $return_path has the same value as $sender_address, but if, for example, an incoming message to a mailing list has been expanded by a director that specifies a specific address for delivery error messages, $return_path contains the new error address, while $sender_address contains the original sender address that was received with the message.

$return_size_limit
This contains the value set in the `return_size_limit` option, rounded up to a multiple of 1,000.

$route_option
A router may set up an arbitrary string to be passed to a transport via this variable. Currently, only the **queryprogram** router has the ability to do so.

$self_hostname

The router option `self` can be set to the values `local` or `pass` (among others). These cause the address to be passed over to the directors, as if its domain were a local domain, or to be passed on to the next router, respectively. While subsequently directing or routing (and doing any deliveries), $self_hostname is set to the name of the local host that the router encountered.

$sender_address

When a message is being processed, this variable contains the sender's address that was received in the message's envelope. For delivery failure reports, the value of this variable is the empty string.

$sender_address_domain

This variable holds the domain portion of $sender_address.

$sender_address_local_part

This variable holds the local part portion of $sender_address.

$sender_fullhost

When a message has been received from a remote host, this variable contains the hostname and IP address in a single string, which always ends with the IP address in square brackets. The format of the rest of the string depends on whether the host issued a `HELO` or `EHLO` SMTP command, and whether the hostname was verified by looking up its IP address. (Looking up the IP address can be forced by the `host_lookup` option, independent of verification.) A plain hostname at the start of the string is a verified hostname; if this is not present, verification either failed or was not requested. A hostname in parentheses is the argument of a `HELO` or `EHLO` command. This is omitted if it is identical to the verified hostname or to the host's IP address in square brackets.

$sender_helo_name

When a message has been received from a remote host that has issued a `HELO` or `EHLO` command, the first item in the argument of that command is placed in this variable. It is also set if `HELO` or `EHLO` is used when a message is received using SMTP locally via the *-bs* or *-bS* options.

$sender_host_address

When a message has been received from a remote host, this variable contains the host's IP address.

$sender_host_authenticated

During message delivery, this variable contains the name (not the public name) of the authenticator driver that successfully authenticated the client from which the message was received. It is empty if there was no successful authentication.

$sender_host_name

When a message has been received from a remote host, this variable contains the host's name as verified by looking up its IP address. If verification failed or was not requested, this variable contains the empty string.

$sender_host_port

When a message is received from a remote host, this variable contains the port number that was used on the remote host.

$sender_ident

When a message has been received from a remote host, this variable contains the identification received in response to an RFC 1413 request. When a message has been received locally, this variable contains the login name of the user that called Exim.

$sender_rcvhost

This variable is provided specifically for use in *Received:* header lines. It starts with either the verified hostname (as obtained from a reverse DNS lookup), or, if there is no verified hostname, the IP address in square brackets. After that there may be text in parentheses. When the first item is a verified hostname, the first thing in the parentheses is the IP address in square brackets. There may also be items of the form `helo=`*xxxx* if `HELO` or `EHLO` was used and its argument was not identical to the real hostname or IP address, and `ident=`*xxxx* if an RFC 1413 ident string is available. If all three items are present in the parentheses, a newline and tab are inserted into the string to improve the formatting of the *Received:* header.

$sn0 to $sn9

These variables are copies of the values of the $n0 to $n9 accumulators that were current at the end of the system filter file. This allows a system filter file to set values that can be tested in users' filter files. For example, a system filter could set a value indicating how likely it is that a message is junk mail.

$spool_directory

This variable holds the name of Exim's spool directory.

$thisaddress

This variable is set only during the processing of the *foranyaddress* command in a filter file. Its use is explained in the description of that command in the section "Testing a List of Addresses," in Chapter 10, *Message Filtering*.

$tls_cipher

When a message is received from a remote host over an encrypted SMTP connection, this variable is set to the cipher that was negotiated.

$tls_peerdn

When a message is received from a remote host over an encrypted SMTP connection, and Exim is configured to request a certificate from the client, this variable is set to the value of the Distinguished Name of the certificate.

$tod_bsdinbox

This variable holds the date and time of day, in the format required for BSD-style mailbox files; for example: `Thu Oct 17 17:14:09 1995`.

$tod_full

This variable holds a full version of the date and time; for example: `Wed, 16 Oct 1995 09:51:40 +0100`. The time zone is always given as a numerical offset from GMT.

$tod_log

This variable holds the date and time in the format used for writing Exim's log files; for example: `1995-10-12 15:32:29`.

$value

This variable contains the result of an expansion lookup operation during the expansion of the "success" substring. Also, if a **domainlist** router has a lookup pattern in a route item, $value contains the data that was looked up during the expansion of the host list.

$version_number

This variable holds the version number of Exim.

$warnmsg_delay

This variable is set only during the creation of a message warning about a delivery delay. Details of its use are explained in the section "Customizing Warning Messages," in Chapter 19, *Miscellany*.

$warnmsg_recipients

This variable is set only during the creation of a message warning about a delivery delay. Details of its use are explained in the section "Customizing Warning Messages," in Chapter 19.

B
Regular Expressions

Regular expression support in Exim is provided by the PCRE library, which implements regular expressions whose syntax and semantics are the same as those in Perl.* The description here is taken from the PCRE documentation, and is intended as reference material. For an introduction to regular expressions, see *Mastering Regular Expressions* by Jeffrey Friedl (O'Reilly).

When you use a regular expression in an Exim configuration, you have to be a little careful about backslash, dollar, and brace characters, which quite often appear in regular expressions, because these characters are also interpreted specially by Exim. Backslash is special inside quoted strings, and all four characters are special in a string that is expanded. One way of setting up such configuration items is as follows:

- First of all, create your regular expression according to the description in this appendix. In other words, find the character string that you ultimately want to pass to the regular expression matcher.
- If the Exim option you are setting is one that is expanded, go through your expression and insert a backslash before every backslash, dollar, and brace character.
- If the Exim option is a string inside double quotes, go through the expression again, inserting a backslash before every backslash.

* PCRE was implemented and is maintained by the same author as Exim. Although originally written in support of Exim, it is a freestanding library that is now used in many other programs. However, the version that is incorporated in the Exim source is minimal. If you want to use PCRE in other programs, you should obtain and install the full distribution from *ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre*.

If you are using Exim Version 3.14 or later, you do not need to use double quotes unless you specifically need to use escape sequences in the string. For example, suppose you want to allow relaying to any domain whose first component consists of letters followed by three digits within some enclosing domain. A regular expression that matches the required domains is:

```
^(?>[a-z]+)\d{3}\.enc\.example$
```

You could use this regular expression to control relaying with the setting:

```
relay_domains = ^(?>[a-z]+)\d{3}\.enc\.example$
```

because the value of `relay_domains` is not expanded. However, if you wanted to use the same pattern in a `domains` option in a director or router, you would have to set it as:

```
domains = ^(?>[a-z]+)\\d\{3\}\\.enc\\.example\$
```

because this option *is* expanded before it is used. If for some reason you choose to enclose the string in quotes, it has to be:

```
domains = "^(?>[a-z]+)\\\\d\\{3\\}\\\\.enc\\\\.example\\$"
```

which is one reason for avoiding the use of quotes if possible.

Testing Regular Expressions

The PCRE library comes with a program called *pcretest* that can be used to test regular expressions, though it was originally written to test the library itself. The Exim distribution includes *pcretest*, but it does not install it automatically. If you have built Exim from source, you will find *pcretest* in the *util* directory.

When you run *pcretest*, it prompts for a regular expression, which must be supplied between delimiters and can be followed by flags. Then it prompts for a succession of data lines to be matched; for each one, the results of the match are output. For example:

```
$ pcretest
PCRE version 3.4 22-Aug-2000

 re> /^abc(\d+)/i
data> aBc123xyz
 0: aBc123
 1: 123
data> xyz
No match
```

After a successful match, string 0 is what the entire pattern matched, and strings 1, 2, and so on are the contents of the captured substrings. For more details of *pcretest*, take a look at its specfication, which is supplied in the file *doc/pcretest.txt* in the Exim distribution.

Metacharacters

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern:

```
The quick brown fox
```

matches a portion of a subject string that is identical to itself. The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of *metacharacters*, which do not stand for themselves, but instead are interpreted in some special way. When a pattern matches, it is possible to arrange for portions of the subject string that matched particular parts of the pattern to be identified. These are called *captured substrings*.

There are two different sets of metacharacters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized in square brackets. Outside square brackets, the metacharacters are as follows:

`\`	general escape character with several uses
`^`	Assert start of subject (or line, in multiline mode)
`$`	Assert end of subject (or line, in multiline mode)
`.`	Match any character except newline (by default)
`[`	Start character class definition
`|`	Start of alternative branch
`(`	Start subpattern
`)`	End subpattern
`?`	Extends the meaning of (
	also 0 or 1 quantifier
	also quantifier minimizer
`*`	0 or more quantifier
`+`	1 or more quantifier
`{`	Start minimum/maximum quantifier

Part of a pattern that is in square brackets is called a *character class*. In a character class, the only metacharacters are as follows:

`\`	General escape character
`^`	Negate the class, if the first character
`-`	Indicates character range
`]`	Terminates the character class

The following sections describe the use of each of the metacharacters.

Backslash

The backslash character has several uses. First, if it is followed by a nonalphanumeric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, if you want to match a `*` character, write `\*` in the pattern. The effect of backslash applies whether or not the following character would otherwise be interpreted as a metacharacter, so it is always safe to precede a nonalphanumeric with a backslash to specify that it stands for itself. In particular, if you want to match a backslash, write `\\`.

If a pattern contains the `(?x)` option (see later in this appendix), whitespace in the pattern (other than in a character class) and characters between a `#` outside a character class and the next newline character are ignored. An escaping backslash can be used to include whitespace or a `#` character as part of the pattern in this circumstance.

A second use of backslash provides a way of encoding nonprinting characters in patterns in a visible manner. There is no restriction on the appearance of nonprinting characters, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

`\a`	Alarm, that is, the BEL character (character 7)
`\cx`	"Control-x," where x is any character
`\e`	Escape (character 27)
`\f`	Formfeed (character 12)
`\n`	Newline or linefeed (character 10)
`\r`	Carriage return (character 13)
`\t`	Tab (character 9)
`\xhh`	Character with hex code hh
`\ddd`	Character with octal code ddd, or a backreference (see later in this appendix)

The precise effect of `\cx` is as follows: if `x` is a lowercase letter, it is converted to uppercase. Then bit 6 of the character (hex 40) is inverted. Thus `\cz` becomes hex `1A`, but `\c{` becomes hex `3B`, and `\c;` becomes hex `7B`.

After `\x`, up to two hexadecimal digits are read (letters can be in upper- or lowercase).

After `\0`, up to two further octal digits are read. In both cases, if there are fewer than two digits, just those that are present are used. Thus, the sequence `\0\x\07` specifies two binary zeros followed by a BEL character. Make sure you supply two digits after the initial zero if the character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a *back reference*. A description of how this works is given later in the section "Back References," following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE rereads up to three octal digits following the backslash, and generates a single byte from the least significant 8 bits of the value. Any subsequent digits stand for themselves. For example:

`\040`	Another way of writing a space
`\40`	The same, provided there are fewer than 40 previous capturing subpatterns
`\7`	Always a back reference
`\11`	Might be a back reference, or another way of writing a tab
`\011`	Always a tab
`\0113`	A tab followed by the character `3`
`\113`	The character with octal code 113 (since there can be no more than 99 back references)
`\377`	A byte with every bit set to 1
`\81`	Either a back reference, or a binary zero followed by the two characters `8` and `1`

Note that octal values of 100 or greater must not be introduced by a leading zero, because no more than three octal digits are ever read.

All the sequences that define a single byte value can be used both inside and outside character classes. In addition, inside a character class, the sequence `\b` is interpreted as the backspace character (character 8). Outside a character class, it has a different meaning (see later in this appendix).

The third use of backslash is for specifying generic character types:

`\d`	any decimal digit
`\D`	Any character that is not a decimal digit
`\s`	Any whitespace character
`\S`	Any character that is not a whitespace character
`\w`	Any "word" character
`\W`	Any "nonword" character

Each pair of escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.

A "word" character is any letter or digit or the underscore character; that is, any character that can be part of a Perl "word."

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, because there is no character to match.

The fourth use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match, without consuming any characters from the subject string. The use of subpatterns for more complicated assertions is described later in the section "Assertions." The backslashed assertions are:

`\b`	Word boundary
`\B`	Not a word boundary
`\A`	Start of subject (independent of multiline mode)
`\Z`	End of subject or newline at end (independent of multiline mode)
`\z`	End of subject (independent of multiline mode)

These assertions may not appear in character classes (but note that `\b` has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the previous character do not both match `\w` or `\W` (that is, one matches `\w` and the other matches `\W`), or the start or end of the string if the first or last character matches `\w`, respectively.

The `\A`, `\Z`, and `\z` assertions differ from the traditional circumflex and dollar (see the section "Circumflex and Dollar," later in this appendix) in that they only ever match at the very start and end of the subject string, whatever options are set. The difference between `\Z` and `\z` is that `\Z` matches before a newline that is the last character of the string as well as at the end of the string, whereas `\z` matches only at the end.

Changing Matching Options

Some details of the matching process are controlled by options that can be changed from within the pattern itself. The syntax is a sequence of Perl option letters enclosed between `(?` and `)`. The option letters are:

`i`	Case-independent matching
`m`	"Multiline" matching (see the section "Circumflex and Dollar")
`s`	"Single-line" matching (see the section "Dot (Period, Full Stop)")
`x`	Ignore literal whitespace in the pattern

For example, `(?im)` sets caseless, multiline matching. It is also possible to unset these options by preceding the letter with a hyphen; a combined setting and unsetting such as `(?im-sx)` is also permitted. If a letter appears both before and after the hyphen, the option is unset.

The scope of these option changes depends on where in the pattern the setting occurs. For settings that are outside any parenthesized subpattern (defined later in this appendix), the effect is the same as if the options were set or unset at the start of matching. The following patterns all behave in exactly the same way:

```
(?i)abc
a(?i)bc
ab(?i)c
abc(?i)
```

In other words, such "top level" settings apply to the whole pattern (unless there are other changes inside subpatterns). If there is more than one setting of the same option at top level, the rightmost setting is used. If an option change occurs inside a subpattern, the effect is different. Such a change affects only that part of the subpattern that follows it, so:

```
(a(?i)b)c
```

matches `abc` and `aBc` and no other strings. By this means, options can be made to have different settings in different parts of the pattern. Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example:

```
(a(?i)b|c)
```

matches `ab`, `aB`, `c`, and `C`, even though when matching `C`, the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behavior otherwise.

In addition to the standard Perl options, PCRE has some extra ones of its own. These are as follows:

`U`	Invert greedy/ungreedy matching
`R`	Recursive matching

Their use is described later in this appendix, in the section "Repetition," and the section "Recursive Patterns," respectively.

Circumflex and Dollar

Outside a character class, in the default matching mode, the circumflex character is an assertion that is true only if the current matching point is at the start of the subject string. Inside a character class, circumflex has an entirely different meaning (see the section "Square Brackets," later in this appendix).

Circumflex is used in Exim configuration files to indicate that the string it introduces is a regular expression rather than a literal string, and it is interpreted as part of that expression. However, when a string can only be a regular expression (for example, as part of the `matches` condition in a string expansion), a leading circumflex is not necessary.

If all possible alternatives start with a circumflex (that is, if the pattern is constrained to match only at the start of the subject), it is said to be an "anchored" pattern. (There are also other constructs that can cause a pattern to be anchored.)

A dollar character is an assertion that is true only if the current matching point is at the end of the subject string, or immediately before a newline character that is the last character in the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class.

The meanings of the circumflex and dollar characters are changed if the `(?m)` option is set. This is referred to in Perl as the "multiline" option. When this is the case, they match immediately after and immediately before an internal newline character, respectively, in addition to matching at the start and end of the subject string. For example, the pattern `^abc$` matches the subject string `def\nabc` (where `\n` represents a newline) in multiline mode, but not otherwise. Consequently, patterns that are anchored in single-line mode because all branches start with `^` are not anchored in multiline mode.

Note that the sequences `\A`, `\Z`, and `\z` can be used to match the start and end of the subject in both modes, and if all branches of a pattern start with `\A`, the pattern is always anchored.

Dot (Period, Full Stop)

Outside a character class, a dot in the pattern matches any one character in the subject, including a nonprinting character, but not (by default) newline. If the `(?s)` option is set, dots match newlines as well. (In Perl, this is referred to as the "single-line" option.) The handling of dot is entirely independent of the handling of circumflex and dollar, the only relationship being that they both involve newline characters. Dot has no special meaning in a character class.

Square Brackets

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special. If a closing square bracket is required as a member of the class, it should be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject; the character must be in the set of characters defined by the class, unless the first character in the class is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class [aeiou] matches any lowercase vowel, while [^aeiou] matches any character that is not a lowercase vowel. This use of circumflex is just a convenient notation for specifying the characters that are in the class by enumerating those that are not. A character class that starts with a circumflex is not an assertion: it still consumes a character from the subject string, and fails if the current pointer is at the end of the string.

When caseless matching is set, any letters in a class represent both their uppercase and lowercase versions, so for example, a caseless [aeiou] matches U as well as u, and a caseless [^aeiou] does not match U, whereas a caseful version would.

The newline character is never treated in any special way in character classes, whatever the setting of the (?s) or (?m) options is. A class such as [^a] always matches a newline.

The hyphen (minus) character can be used to specify a range of characters in a character class. For example, [d-m] matches any letter between d and m, inclusive. If a hyphen is required in a class, it must be escaped with a backslash, or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

It is not possible to have the literal character] as the end character of a range. A pattern such as [W-]46] is interpreted as a class of two characters (W and -) followed by a literal string 46], so it would match W46] or -46]. However, if the] is escaped with a backslash, it is interpreted as the end of range, so [W-\]46] is interpreted as a single class containing a range followed by two separate characters. The octal or hexadecimal representation of] can also be used to end a range.

Ranges operate in ASCII-collating sequence. They can also be used for characters specified numerically; for example [\000-\037]. If a range that includes letters is used when caseless matching is set, it matches the letters in either case. For example, [W-c] is equivalent to [][\^_`wxyzabc], matched caselessly.

The character types \d, \D, \s, \S, \w, and \W may also appear in a character class, and add the characters that they match to the class. For example, [\dABCDEF] matches any hexadecimal digit.

The interpretation of a nonnegated class can be understood by reading it with the word "or" between each item, whereas for a negated class, "and not" is implied. For example, `[ANZ]` matches a character that is `A`, `N`, or `Z`, whereas `[^ANZ]` matches a character that is not `A`, `N`, and `Z`. This means that a negated class can conveniently be used with the uppercase character types to specify a more restricted set of characters than the matching lowercase type. For example, the class `[^\W_]` matches any letter or digit, but not underscore, because it matches a character that is not a nonword character (that is, it *is* a word character), and not an underscore.

All nonalphanumeric characters other than `\`, `-`, `^` (at the start) and the terminating `]` are nonspecial in character classes, but it does no harm if they are escaped.

POSIX Character Classes

Perl 5.6 supports POSIX notation for character classes, which uses names enclosed by `[:` and `:]` within the enclosing square brackets. PCRE supports this notation. For example:

```
[01[:alpha:]%]
```

matches `0`, `1`, any alphabetic character, or `%`. The supported class names are:

`alnum`	letters and digits
`alpha`	Letters
`ascii`	Character codes 0–127
`cntrl`	Control characters
`digit`	Decimal digits (same as `\d`)
`graph`	Printing characters, excluding space
`lower`	Lowercase letters
`print`	Printing characters, including space
`punct`	Printing characters, excluding letters and digits
`space`	Whitespace (same as `\s`)
`upper`	Uppercase letters
`word`	"Word" characters (same as `\w`)
`xdigit`	Hexadecimal digits

The names `ascii` and `word` are Perl extensions. Another Perl extension is negation, which is indicated by a `^` character after the colon. For example:

```
[12[:^digit:]]
```

matches `1`, `2`, or any nondigit. PCRE and Perl also recognize the POSIX syntax `[.ch.]` and `[=ch=]` where "ch" is a "collating element," but these are not supported, and an error is given if they are encountered.

Vertical Bar

Vertical bar characters are used to separate alternative patterns. For example, the following pattern:

```
gilbert|sullivan
```

matches either `gilbert` or `sullivan`. Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined in the next section), "succeeds" means matching the rest of the main pattern as well as the alternative in the subpattern.

Subpatterns

Subpatterns are delimited by parentheses (round brackets), which can be nested. Marking part of a pattern as a subpattern does two things:

- It localizes a set of alternatives. For example, the pattern:

  ```
  cat(aract|erpillar|)
  ```

  matches one of the words `cat`, `cataract`, or `caterpillar`. Without the parentheses, it would match `cataract`, `erpillar`, or the empty string.

- It sets up the subpattern as a capturing subpattern. When the whole pattern matches, that portion of the subject string that matched the subpattern is passed back to the caller, and in Exim, such values are made available in the numerical variables $1, $2, and so on. Opening parentheses are counted from left to right (starting from 1) to obtain the numbers of the capturing subpatterns.

For example, if the string `the red king` is matched against the following pattern:

```
the ((red|white) (king|queen))
```

the captured substrings are `red king`, `red`, and `king`, and are numbered 1, 2, and 3, respectively.

The fact that plain parentheses fulfill two functions is not always helpful. There are often times when a grouping subpattern is required without a capturing requirement. If an opening parenthesis is followed by `?:`, the subpattern does not do any capturing, and is not counted when computing the number of any subsequent capturing subpatterns. For example, if the string "the white queen" is matched against the pattern:

```
the ((?:red|white) (king|queen))
```

the captured substrings are `white queen` and `queen`, and are numbered 1 and 2. The maximum number of captured substrings is 99, and the maximum number of all subpatterns, both capturing and noncapturing, is 200.

As a convenient shorthand, if any option settings are required at the start of a noncapturing subpattern, the option letters may appear between the `?` and the `:`. Thus, the two patterns:

```
(?i:saturday|sunday)
(?:(?i)saturday|sunday)
```

match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the subpattern is reached, an option setting in one branch does affect subsequent branches, so the previous patterns match `SUNDAY` as well as `Saturday`, and any other case variants.

Repetition

Repetition is specified by quantifiers, which can follow any of the following items:

- A single character, possibly escaped
- The . metacharacter
- A character class
- A back reference (see the next section)
- A parenthesized subpattern (unless it is an assertion; see later in this appendix)

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma. The numbers must be less than 65536, and the first must be less than or equal to the second. For example:

```
z{2,4}
```

matches `zz`, `zzz`, or `zzzz`. A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus:

```
[aeiou]{3,}
```

matches at least three successive vowels, but may match many more, while the following:

```
\d{8}
```

matches exactly eight digits. An opening curly bracket that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, `{,6}` is not a quantifier, but a literal string of four characters.

The quantifier `{0}` is permitted, causing the expression to behave as if the previous item and the quantifier were not present.

For convenience (and historical compatibility), the three most common quantifiers have single-character abbreviations:

`*`	Equivalent to {0,}
`+`	Equivalent to {1,}
`?`	Equivalent to {0,1}

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

```
(a?)*
```

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does, in fact, match no characters, the loop is forcibly broken.

By default, the quantifiers are "greedy"; that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between the sequences /* and */, within the sequence, individual * and / characters may appear. An attempt to match C comments by applying the pattern:

```
/\*.*\*/
```

to the string:

```
/* first comment */  not comment  /* second comment */
```

fails, because it matches the entire string owing to the greediness of the `.*` item. However, if a quantifier is followed by a question mark, it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern:

```
/\*.*?\*/
```

does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in:

```
\d??\d
```

which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the `(?U)` option is set (an option that is not available in Perl), the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behavior.

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1, or with a limited maximum, more store is required for the compiled pattern, in proportion to the size of the minimum or maximum.

If a pattern starts with `.*` (or `.{0,}`) and the `(?s)`option is set, thus allowing the dot to match newlines, the pattern is implicitly anchored, because whatever follows will be tried against every character position in the subject string. At first, the `.*` item consumes the entire subject string, but if the rest of the pattern does not match, it "gives up" characters one by one until either the whole pattern does match, or the start of the string is reached (when `.*` matches no characters). If `.*?` is used instead of `.*`, the same thing happens, but in the opposite order.

For such patterns, therefore, there is no point in retrying the overall match at any position after the first. PCRE treats such a pattern as though it were preceded by `\A`. In cases where it is known that the subject string contains no newlines, it is worth setting `(?s)` when the pattern begins with `.*` in order to obtain this optimization, or alternatively using ^ to indicate anchoring explicitly.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after the following:

```
(tweedle[dume]{3}\s*)+
```

has matched `tweedledum tweedledee`, the value of the captured substring is `tweedledee`. However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations, and they retain the last values that were set. For example, after the following:

```
/(a|(b))+/
```

matches `aba`, the value of the second captured substring is `b`.

Back References

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (that is, to its left) in the pattern, provided there have been that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. See earlier in the section "Backslash" for further details of the handling of digits following a backslash.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself. So the pattern:

```
(sens|respons)e and \1ibility
```

matches `sense and sensibility` and `response and responsibility`, but not `sense and responsibility`. If caseful matching is in force at the time of the back reference, the case of letters is relevant. For example:

```
((?i)rah)\s+\1
```

matches `rah rah` and `RAH RAH`, but not `RAH rah`, even though the original capturing subpattern is matched caselessly.

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, any back references to it always fail. For example, the pattern:

```
(a|(bc))\2
```

always fails if it starts to match `a` rather than `bc`. Because there may be up to 99 back references, all digits following the backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, some delimiter must be used to terminate the back reference. If the `(?x)` option is set, this can be whitespace. Otherwise an empty comment can be used.

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, `(a\1)` never matches. However, such references can be useful inside repeated subpatterns. For instance, the pattern:

```
(a|b\1)+
```

matches any number of `a`s and also `aba`, `ababbaa`, and so on. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the earlier example, or by a quantifier with a minimum of zero.

Assertions

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as `\b`, `\B`, `\A`, `\Z`, `\z`, `^` and `$` are described earlier in the the section "Backslash." More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it.

An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed. Lookahead assertions start with `(?=` for positive assertions and `(?!` for negative assertions. For example:

```
\w+(?=;)
```

matches a word followed by a semicolon, but does not include the semicolon in the match, and the following:

```
foo(?!bar)
```

matches any occurrence of `foo` that is not followed by `bar`. Note that the apparently similar pattern:

```
(?!foo)bar
```

does not find an occurrence of `bar` that is preceded by something other than `foo`; it finds any occurrence of `bar` whatsoever, because the assertion `(?!foo)` is always true when the next three characters are `bar`. A lookbehind assertion is needed to achieve this effect.

Lookbehind assertions start with `(?<=` for positive assertions and `(?<!` for negative assertions. For example:

```
(?<!foo)bar
```

does find an occurrence of `bar` that is not preceded by `foo`. The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. The only permitted repetition is a quantifier with a fixed value; for example, `a{4}`; unlimited repeats are forbidden. However, if there are several alternatives in a lookbehind assertion, they do not all have to have the same fixed length. Thus:

```
(?<=bullock|donkey)
```

is permitted, but:

```
(?<!dogs?|cats?)
```

causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl 5.005, which requires all branches to match the same length of string. An assertion such as:

```
(?<=ab(c|de))
```

is not permitted, because its single top-level branch can match two different lengths, but it is acceptable if rewritten to use two top-level branches:

```
(?<=abc|abde)
```

Similarly, the previous example could be rewritten as:

```
(?<!dog|cat|dogs|cats)
```

The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed number of characters, and then try to match. If there are insufficient characters before the current position, the match fails. Lookbehinds in conjunction with once-only subpatterns can be particularly useful for matching at the ends of strings; an example is given at the end of the section on once-only subpatterns.

Several assertions (of any sort) may occur in succession. For example:

```
(?<=\d{3})(?<!999)foo
```

matches `foo` preceded by three digits that are not `999`. Notice that each of the assertions is applied independently at the same point in the subject string. First, there is a check that the previous three characters are all digits, then there is a check that the same three characters are not `999`. This pattern does *not* match `foo` preceded by six characters, the first of which are digits and the last three of which are not `999`. For example, it doesn't match `123abcfoo`. A pattern to do that is as follows:

```
(?<=\d{3}...)(?<!999)foo
```

This time, the first assertion looks at the preceding six characters, checking that the first three are digits. Then the second assertion checks that the preceding three characters are not `999`.

Assertions can be nested in any combination. For example:

```
(?<=(?<!foo)bar)baz
```

matches an occurrence of `baz` that is preceded by `bar`, which, in turn, is not preceded by `foo`, while the following:

```
(?<=\d{3}(?!999)...)foo
```

is another pattern that matches `foo` preceded by three digits and any three characters that are not `999`.

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

Assertions count towards the maximum of 200 parenthesized subpatterns.

Once-Only Subpatterns

With both maximizing and minimizing repetition, failure of what follows normally causes the repeated item to be reevaluated to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it to fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern `\d+foo` when applied to the following subject line:

```
    123456bar
```

After matching all six digits and then failing to match `foo`, the normal action of the matcher is to try again with only five digits matching the `\d+` item, and then with four, and so on, before ultimately failing. Once-only subpatterns provide the means for specifying that once a portion of the pattern has matched, it is not to be reevaluated in this way, so the matcher would give up immediately on failing to match `foo` the first time. The notation is another kind of special parenthesis, starting with `(?>` as in this example:

```
    (?>\d+)bar
```

This kind of parenthesis “locks up” the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the subject string.

Once-only subpatterns are not capturing subpatterns. Simple cases such as the previous example can be thought of as a maximizing repeat that must swallow everything it can. So, while both `\d+` and `\d+?` are prepared to adjust the number of digits they match in order to make the rest of the pattern match, `(?>\d+)` can only match an entire sequence of digits.

This construction can of course contain arbitrarily complicated subpatterns, and it can be nested.

Once-only subpatterns can be used in conjunction with lookbehind assertions to specify efficient matching at the end of the subject string. Consider a simple pattern such as:

```
xyz$
```

when applied to a long string that does not match. Because matching proceeds from left to right, PCRE will look for each `x` in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as:

```
^.*xyz$
```

the initial `.*` matches the entire string at first, but when this fails (because there is no following `x`), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again, the search for `x` covers the entire string, from right to left, so we are no better off. However, if the pattern is written as:

```
^(?>.*)(?<=xyz)
```

there can be no backtracking for the `.*` item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last three characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of a once-only subpattern is the only way to avoid some failing matches taking a very long time indeed. The following pattern:

```
(\D+|<\d+>)*[!?]
```

matches an unlimited number of substrings that either consist of nondigits, or digits enclosed in `<>`, followed by either `!` or `?`. When it matches, it runs quickly. However, if it is applied to the following:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

it takes a long time before reporting failure. This is because the string can be divided between the two repeats in a large number of ways, and all have to be tried.* If the pattern is changed to:

```
((?>\D+)|<\d+>)*[!?]
```

sequences of nondigits cannot be broken, and failure happens quickly.

Conditional Subpatterns

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion or whether a previous capturing subpattern matched or not. The two possible forms of conditional subpattern are:

```
(?(condition)yes-pattern)
(?(condition)yes-pattern|no-pattern)
```

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are two kinds of conditions. If the text between the parentheses consists of a sequence of digits, the condition is satisfied if the capturing subpattern of that number has previously matched. Consider the following pattern, which contains nonsignificant whitespace to make it more readable (assume the `(?x)` option) and is divided it into three parts for ease of discussion:

```
( \( )?    [^()]+    (?(1) \) )
```

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did (that is, if the subject started with an opening parenthesis), the condition is true, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since the no-pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of nonparentheses, optionally enclosed in parentheses.

If the condition is not a sequence of digits, it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern,

* The example used `[!?]` rather than a single character at the end, because both PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is required for a match, and fail early if it is not present in the string.

again containing nonsignificant whitespace, and with the two alternatives on the second line:

```
(?(?=[^a-z]*[a-z])
\d{2}-[a-z]{3}-\d{2}  |  \d{2}-\d{2}-\d{2} )
```

The condition is a positive lookahead assertion that matches an optional sequence of nonletters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise, it is matched against the second. This pattern matches strings in one of the two forms *dd-aaa-dd* or *dd-dd-dd*, where *aaa* are letters and *dd* are digits.

Comments

The sequence `(?#` marks the start of a comment that continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

If the `(?x)` option is set, an unescaped `#` character outside a character class introduces a comment that continues up to the next newline character in the pattern.

Recursive Patterns

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth. Perl 5.6 provides an experimental facility that allows regular expressions to recurse (among other things). It does this by interpolating Perl code in the expression at runtime, and the code can refer to the expression itself. A Perl pattern to solve the parentheses problem can be created in the following manner:

```
$re = qr{\( (?: (?>[^()]+) | (?p{$re}) )* \)}x;
```

The `(?p{ . . . })` item interpolates Perl code at runtime, and, in this case, refers recursively to the pattern in which it appears. Obviously, PCRE cannot support the interpolation of Perl code. Instead, the special item `(?R)` is provided for the specific case of recursion. This PCRE pattern solves the parentheses problem (assume the `(?x)` option is set so that whitespace is ignored):

```
\( ( (?>[^()]+) | (?R) )* \)
```

First, it matches an opening parenthesis. Then, it matches any number of substrings that can either be a sequence of nonparentheses, or a recursive match of the pattern itself (that is, a correctly parenthesized substring). Finally, there is a closing parenthesis.

This particular example pattern contains nested unlimited repeats, and so the use of a once-only subpattern for matching strings of nonparentheses is important when applying the pattern to strings that do not match. For example, when it is applied to the following:

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa()
```

it yields "no match" quickly. However, if a once-only subpattern is not used, the match runs for a very long time indeed because there are so many different ways the + and * repeats can carve up the subject, and all have to be tested before failure can be reported.

The values set for any capturing subpatterns are those from the outermost level of the recursion at which the subpattern value is set. If the previous pattern is matched against the following:

```
(ab(cd)ef)
```

the value for the capturing parentheses is `ef`, which is the last value taken on at the top level. If additional parentheses are added, giving the following:

```
  \( ( ( (?>[^()]+) | (?R) )* ) \)
```

the string they capture is `ab(cd)ef`, the contents of the top level parentheses. If there are more than 15 capturing parentheses in a pattern, PCRE has to obtain extra memory to store data during a recursion. If no memory can be obtained, it saves data for the first 15 capturing parentheses only, as there is no way to give an out-of-memory error from within a recursion.

Performance

Certain items that may appear in patterns are more efficient than others. It is more efficient to use a character class like `[aeiou]` than a set of alternatives such as `(a|e|i|o|u)`. In general, the simplest construction that provides the required behavior is usually the most efficient.

When a pattern begins with `.*` and the `(?s)` option is set, the pattern is implicitly anchored by PCRE, since it can match only at the start of a subject string. However, if `(?s)` is not set, PCRE cannot make this optimization, because the `.` metacharacter does not then match a newline, and if the subject string contains newlines, the pattern may match from the character immediately following one of them, instead of from the very start. For example, the pattern:

```
(.*) second
```

matches the subject `first\nand second` (where `\n` stands for a newline character) with the first captured substring being `and`. In order to do this, PCRE may have to try the match several time, starting after every newline in the subject as well as at the start.

If you are using such a pattern with subject strings that do not contain newlines, the best performance is obtained by setting `(?s)`, or starting the pattern with `^.*` to indicate explicit anchoring. That saves PCRE from having to scan along the subject looking for a newline to restart at.

Beware of patterns that contain nested indefinite repeats. These can take a long time to run when applied to a string that does not match. Consider the pattern fragment:

```
(a+)*
```

This can match `aaaa` in 33 different ways,[*] and this number increases very rapidly as the string gets longer. When the remainder of the pattern is such that the entire match is going to fail, PCRE has, in principle, to try every possible variation, and this can take an extremely long time.

An optimization catches some of the more simple cases such as:

```
(x+)*y
```

where a literal character follows. Before embarking on the standard matching procedure, PCRE checks that there is a `y` later in the subject string, and if there is not, it fails the match immediately. However, when there is no following literal, this optimization cannot be used. You can see the difference by comparing the behavior of:

```
(x+)*\d
```

with the previous pattern. The former gives a failure almost instantly when applied to a whole line of `x` characters, whereas the latter takes an appreciable time with strings longer than about 20 characters.

* The `*` repeat can match 0, 1, 2, 3, or 4 times, and for each of those cases other than 0, the `+` repeat can match different numbers of times.

Index

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com.*

About the Author

Philip Hazel has a Ph.D. in applied mathematics, but has spent the last 30 years writing general-purpose software for the Computing Service at the University of Cambridge in England. Since moving from an IBM mainframe to Unix about ten years ago, he has gotten more and more involved with email. Philip started developing Exim in 1995 and is its sole author.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of *Exim: The Mail Transfer Agent* is an aye-aye. The aye-aye is part of the order of primates, and in fact is part of the lemur group. Native to Madagascar, they are considered one of the strangest looking primates, and not very much is known about them. A full-grown adult is about the size of a raccoon. Its features include large round ears, black fur with white spots, a flat nose, and big round eyes. Two very distinctive characteristics of the aye-aye are its incisor teeth, which never stop growing, and its long, spindly fingers, of which the middle finger is the longest. Both of these traits are used as tools in hunting food. The aye-aye lives mostly on bug larvae and fruit; it often uses its teeth to break open dead tree bark, then uses its long middle finger to reach inside and take hold of the bugs.

The aye-aye is completely nocturnal, and lives mostly in trees in the forest. Unfortunately, it is dangerously close to extinction. One reason for this is that its natural habitat, the rain forest, is gradually being destroyed for resources. Due to this loss of its food source, the aye-aye has had to forage for food in other areas, and often steals from local farms. For this reason, it is killed as a pest. Also, in parts of Madagascar, there is a superstition that the aye-aye is a harbinger of bad luck and death; therefore, it is often killed on sight. However, steps are being taken to ensure the safety of the species, such as breeding some in captivity and declaring certain areas of the forest as protected.

Mary Brady was the production editor and proofreader and Mark Nigara was the copyeditor for *Exim: The Mail Transfer Agent.* Jane Ellin and Claire Cloutier provided quality control. Ann Schirmer, Gabe Weiss, and Lucy Muellner provided production assistance. Nancy Crumpton wrote the index.

Ellie Volckhausen designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Erica Corwell produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout based on a series design by Nancy Priest. The print version of this book was created by translating the DocBook XML markup of its source files into a set of gtroff macros using a filter developed at O'Reilly & Associates by Norman Walsh. Steve Talbott designed and wrote the underlying macro set on the basis of the GNU *troff –gs* macros; Lenny Muellner adapted them to XML and implemented the book design. The GNU groff text formatter Version 1.11.1 was used to generate PostScript output. The text and heading fonts are ITC Garamond Light and Garamond Book; the code font is Constant Willison. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. This colophon was written by Mary Brady.

Whenever possible, our books use a durable and flexible lay-flat binding. If the page count exceeds this binding's limit, perfect binding is used.