Specification of the Exim Mail Transfer Agent

Philip Hazel

University of Cambridge Computing Service

Specification of the Exim Mail Transfer Agent

Author: Philip Hazel

University of Cambridge Computing Service
New Museums Site, Pembroke Street, Cambridge CB2 3QH, England

Copyright © 2007 University of Cambridge

Revision 4.68 23 August 2007

Contents

1. Introduction **1**

1.1 Exim documentation 1
1.2 FTP and web sites 2
1.3 Mailing lists 2
1.4 Exim training 3
1.5 Bug reports 3
1.6 Where to find the Exim distribution 3
1.7 Limitations 3
1.8 Run time configuration 4
1.9 Calling interface 4
1.10 Terminology 4

2. Incorporated code **6**

3. How Exim receives and delivers mail **8**

3.1 Overall philosophy 8
3.2 Policy control 8
3.3 User filters 8
3.4 Message identification 9
3.5 Receiving mail 9
3.6 Handling an incoming message 10
3.7 Life of a message 10
3.8 Processing an address for delivery 11
3.9 Processing an address for verification 12
3.10 Running an individual router 12
3.11 Duplicate addresses 13
3.12 Router preconditions 13
3.13 Delivery in detail 14
3.14 Retry mechanism 15
3.15 Temporary delivery failure 15
3.16 Permanent delivery failure 15
3.17 Failures to deliver bounce messages 16

4. Building and installing Exim **17**

4.1 Unpacking 17
4.2 Multiple machine architectures and operating systems 17
4.3 DBM libraries 17
4.4 Pre-building configuration 18
4.5 Support for iconv() 19
4.6 Including TLS/SSL encryption support 19
4.7 Use of tcpwrappers 20
4.8 Including support for IPv6 20
4.9 The building process 20
4.10 Output from “make” 21
4.11 Overriding build-time options for Exim 21
4.12 OS-specific header files 22
4.13 Overriding build-time options for the monitor 23
4.14 Installing Exim binaries and scripts 23
4.15 Installing info documentation 24
4.16 Setting up the spool directory 24
4.17 Testing 24
4.18 Replacing another MTA with Exim 25

4.19 Upgrading Exim 26
4.20 Stopping the Exim daemon on Solaris 26

5. The Exim command line 27

5.1 Setting options by program name 27
5.2 Trusted and admin users 27
5.3 Command line options 28

6. The Exim run time configuration file 48

6.1 Using a different configuration file 48
6.2 Configuration file format 49
6.3 File inclusions in the configuration file 49
6.4 Macros in the configuration file 50
6.5 Macro substitution 50
6.6 Redefining macros 50
6.7 Overriding macro values 51
6.8 Example of macro usage 51
6.9 Conditional skips in the configuration file 51
6.10 Common option syntax 51
6.11 Boolean options 52
6.12 Integer values 52
6.13 Octal integer values 52
6.14 Fixed point numbers 52
6.15 Time intervals 52
6.16 String values 53
6.17 Expanded strings 53
6.18 User and group names 53
6.19 List construction 54
6.20 Changing list separators 54
6.21 Empty items in lists 54
6.22 Format of driver configurations 55

7. The default configuration file 57

7.1 Main configuration settings 57
7.2 ACL configuration 59
7.3 Router configuration 62
7.4 Transport configuration 65
7.5 Default retry rule 65
7.6 Rewriting configuration 66
7.7 Authenticators configuration 66

8. Regular expressions 67

8.1 Testing regular expressions 67

9. File and database lookups 69

9.1 Examples of different lookup syntax 69
9.2 Lookup types 69
9.3 Single-key lookup types 70
9.4 Query-style lookup types 72
9.5 Temporary errors in lookups 73
9.6 Default values in single-key lookups 73
9.7 Partial matching in single-key lookups 74
9.8 Lookup caching 75

9.9 Quoting lookup data ... 75
9.10 More about dnsdb ... 75
9.11 Pseudo dnsdb record types ... 76
9.12 Multiple dnsdb lookups ... 77
9.13 More about LDAP ... 77
9.14 Format of LDAP queries ... 78
9.15 LDAP quoting ... 78
9.16 LDAP connections ... 79
9.17 LDAP authentication and control information ... 79
9.18 Format of data returned by LDAP ... 81
9.19 More about NIS+ ... 81
9.20 SQL lookups ... 82
9.21 More about MySQL, PostgreSQL, Oracle, and InterBase ... 82
9.22 Specifying the server in the query ... 83
9.23 Special MySQL features ... 83
9.24 Special PostgreSQL features ... 83
9.25 More about SQLite ... 84

10. Domain, host, address, and local part lists ... 85

10.1 Expansion of lists ... 85
10.2 Negated items in lists ... 85
10.3 File names in lists ... 85
10.4 An lsearch file is not an out-of-line list ... 86
10.5 Named lists ... 86
10.6 Named lists compared with macros ... 87
10.7 Named list caching ... 87
10.8 Domain lists ... 88
10.9 Host lists ... 90
10.10 Special host list patterns ... 90
10.11 Host list patterns that match by IP address ... 90
10.12 Host list patterns for single-key lookups by host address ... 91
10.13 Host list patterns that match by host name ... 92
10.14 Behaviour when an IP address or name cannot be found ... 93
10.15 Temporary DNS errors when looking up host information ... 93
10.16 Host list patterns for single-key lookups by host name ... 93
10.17 Host list patterns for query-style lookups ... 94
10.18 Mixing wildcarded host names and addresses in host lists ... 94
10.19 Address lists ... 94
10.20 Case of letters in address lists ... 97
10.21 Local part lists ... 97

11. String expansions ... 98

11.1 Literal text in expanded strings ... 98
11.2 Character escape sequences in expanded strings ... 98
11.3 Testing string expansions ... 98
11.4 Forced expansion failure ... 99
11.5 Expansion items ... 99
11.6 Expansion operators ... 108
11.7 Expansion conditions ... 112
11.8 Combining expansion conditions ... 118
11.9 Expansion variables ... 119

12. Embedded Perl ... 135

12.1 Setting up so Perl can be used ... 135
12.2 Calling Perl subroutines ... 135

12.3 Calling Exim functions from Perl 136
12.4 Use of standard output and error by Perl 136

13. Starting the daemon and the use of network interfaces 137

13.1 Starting a listening daemon 137
13.2 Special IP listening addresses 138
13.3 Overriding local_interfaces and daemon_smtp_ports 138
13.4 Support for the obsolete SSMTP (or SMTPS) protocol 138
13.5 IPv6 address scopes 139
13.6 Disabling IPv6 139
13.7 Examples of starting a listening daemon 139
13.8 Recognizing the local host 140
13.9 Delivering to a remote host 140

14. Main configuration 141

14.1 Miscellaneous 141
14.2 Exim parameters 141
14.3 Privilege controls 141
14.4 Logging 141
14.5 Frozen messages 142
14.6 Data lookups 142
14.7 Message ids 142
14.8 Embedded Perl Startup 142
14.9 Daemon 142
14.10 Resource control 142
14.11 Policy controls 143
14.12 Callout cache 144
14.13 TLS 144
14.14 Local user handling 144
14.15 All incoming messages (SMTP and non-SMTP) 144
14.16 Non-SMTP incoming messages 144
14.17 Incoming SMTP messages 144
14.18 SMTP extensions 145
14.19 Processing messages 145
14.20 System filter 145
14.21 Routing and delivery 146
14.22 Bounce and warning messages 146
14.23 Alphabetical list of main options 146

15. Generic options for routers 186

16. The accept router 199

17. The dnslookup router 200

17.1 Problems with DNS lookups 200
17.2 Private options for dnslookup 200
17.3 Effect of qualify_single and search_parents 202

18. The ipliteral router 204

19. The iplookup router 205

20. The manualroute router 207

20.1 Private options for manualroute 207

20.2 Routing rules in route_list 208
20.3 Routing rules in route_data 209
20.4 Format of the list of hosts 209
20.5 Format of one host item 210
20.6 How the list of hosts is used 210
20.7 How the options are used 211
20.8 Manualroute examples 211

21. The queryprogram router 214

22. The redirect router 216

22.1 Redirection data 216
22.2 Forward files and address verification 216
22.3 Interpreting redirection data 217
22.4 Items in a non-filter redirection list 217
22.5 Redirecting to a local mailbox 217
22.6 Special items in redirection lists 218
22.7 Duplicate addresses 220
22.8 Repeated redirection expansion 220
22.9 Errors in redirection lists 220
22.10 Private options for the redirect router 220

23. Environment for running local transports 228

23.1 Concurrent deliveries 228
23.2 Uids and gids 228
23.3 Current and home directories 229
23.4 Expansion variables derived from the address 229

24. Generic options for transports 230

25. Address batching in local transports 236

26. The appendfile transport 238

26.1 The file and directory options 238
26.2 Private options for appendfile 239
26.3 Operational details for appending 248
26.4 Operational details for delivery to a new file 250
26.5 Maildir delivery 250
26.6 Using tags to record message sizes 251
26.7 Using a maildirsize file 251
26.8 Mailstore delivery 252
26.9 Non-special new file delivery 252

27. The autoreply transport 253

27.1 Private options for autoreply 253

28. The lmtp transport 256

29. The pipe transport 258

29.1 Concurrent delivery 258
29.2 Returned status and data 258
29.3 How the command is run 259
29.4 Environment variables 259

29.5 Private options for pipe ... 260
29.6 Using an external local delivery agent ... 264

30. The smtp transport ... 266

30.1 Multiple messages on a single connection ... 266
30.2 Use of the $host and $host_address variables ... 266
30.3 Use of $tls_cipher and $tls_peerdn ... 266
30.4 Private options for smtp ... 266
30.5 How the limits for the number of hosts to try are used ... 274

31. Address rewriting ... 276

31.1 Explicitly configured address rewriting ... 276
31.2 When does rewriting happen? ... 276
31.3 Testing the rewriting rules that apply on input ... 277
31.4 Rewriting rules ... 277
31.5 Rewriting patterns ... 278
31.6 Rewriting replacements ... 279
31.7 Rewriting flags ... 279
31.8 Flags specifying which headers and envelope addresses to rewrite ... 279
31.9 The SMTP-time rewriting flag ... 279
31.10 Flags controlling the rewriting process ... 280
31.11 Rewriting examples ... 280

32. Retry configuration ... 282

32.1 Changing retry rules ... 282
32.2 Format of retry rules ... 282
32.3 Choosing which retry rule to use for address errors ... 283
32.4 Choosing which retry rule to use for host and message errors ... 283
32.5 Retry rules for specific errors ... 284
32.6 Retry rules for specified senders ... 285
32.7 Retry parameters ... 286
32.8 Retry rule examples ... 286
32.9 Timeout of retry data ... 287
32.10 Long-term failures ... 287
32.11 Deliveries that work intermittently ... 288

33. SMTP authentication ... 289

33.1 Generic options for authenticators ... 290
33.2 The AUTH parameter on MAIL commands ... 291
33.3 Authentication on an Exim server ... 292
33.4 Testing server authentication ... 293
33.5 Authentication by an Exim client ... 293

34. The plaintext authenticator ... 295

34.1 Plaintext options ... 295
34.2 Using plaintext in a server ... 295
34.3 The PLAIN authentication mechanism ... 295
34.4 The LOGIN authentication mechanism ... 296
34.5 Support for different kinds of authentication ... 297
34.6 Using plaintext in a client ... 297

35. The cram_md5 authenticator ... 299

35.1 Using cram_md5 as a server 299
35.2 Using cram_md5 as a client 299

36. The cyrus_sasl authenticator 301

36.1 Using cyrus_sasl as a server 301

37. The dovecot authenticator 303

38. The spa authenticator 304

38.1 Using spa as a server 304
38.2 Using spa as a client 304

39. Encrypted SMTP connections using TLS/SSL 306

39.1 Support for the legacy “ssmtp” (aka “smtps”) protocol 306
39.2 OpenSSL vs GnuTLS 306
39.3 GnuTLS parameter computation 307
39.4 Requiring specific ciphers in OpenSSL 307
39.5 Requiring specific ciphers or other parameters in GnuTLS 308
39.6 Configuring an Exim server to use TLS 309
39.7 Requesting and verifying client certificates 310
39.8 Revoked certificates 311
39.9 Configuring an Exim client to use TLS 311
39.10 Multiple messages on the same encrypted TCP/IP connection 311
39.11 Certificates and all that 312
39.12 Certificate chains 312
39.13 Self-signed certificates 312

40. Access control lists 314

40.1 Testing ACLs 314
40.2 Specifying when ACLs are used 314
40.3 The non-SMTP ACLs 315
40.4 The SMTP connect ACL 315
40.5 The EHLO/HELO ACL 315
40.6 The DATA ACLs 315
40.7 The SMTP MIME ACL 316
40.8 The QUIT ACL 316
40.9 The not-QUIT ACL 316
40.10 Finding an ACL to use 317
40.11 ACL return codes 317
40.12 Unset ACL options 318
40.13 Data for message ACLs 318
40.14 Data for non-message ACLs 318
40.15 Format of an ACL 319
40.16 ACL verbs 319
40.17 ACL variables 321
40.18 Condition and modifier processing 321
40.19 ACL modifiers 322
40.20 Use of the control modifier 326
40.21 Summary of message fixup control 328
40.22 Adding header lines in ACLs 328
40.23 ACL conditions 330
40.24 Using DNS lists 334
40.25 Specifying the IP address for a DNS list lookup 334
40.26 DNS lists keyed on domain names 334

40.27 Multiple explicit keys for a DNS list 335
40.28 Data returned by DNS lists 336
40.29 Variables set from DNS lists 336
40.30 Additional matching conditions for DNS lists 336
40.31 Negated DNS matching conditions 337
40.32 Handling multiple DNS records from a DNS list 338
40.33 Detailed information from merged DNS lists 338
40.34 DNS lists and IPv6 339
40.35 Rate limiting incoming messages 339
40.36 Ratelimit options for what is being measured 340
40.37 Ratelimit options for handling fast clients 340
40.38 Using rate limiting 341
40.39 Reading ratelimit data without updating 342
40.40 Address verification 342
40.41 Callout verification 343
40.42 Additional parameters for callouts 344
40.43 Callout caching 346
40.44 Sender address verification reporting 346
40.45 Redirection while verifying 347
40.46 Client SMTP authorization (CSA) 347
40.47 Bounce address tag validation 348
40.48 Using an ACL to control relaying 349
40.49 Checking a relay configuration 350

41. Content scanning at ACL time 351

41.1 Scanning for viruses 351
41.2 Scanning with SpamAssassin 354
41.3 Calling SpamAssassin from an Exim ACL 354
41.4 Scanning MIME parts 356
41.5 Scanning with regular expressions 358
41.6 The demime condition 359

42. Adding a local scan function to Exim 361

42.1 Building Exim to use a local scan function 361
42.2 API for local_scan() 361
42.3 Configuration options for local_scan() 362
42.4 Available Exim variables 363
42.5 Structure of header lines 365
42.6 Structure of recipient items 365
42.7 Available Exim functions 366
42.8 More about Exim's memory handling 370

43. System-wide message filtering 371

43.1 Specifying a system filter 371
43.2 Testing a system filter 371
43.3 Contents of a system filter 371
43.4 Additional variable for system filters 372
43.5 Defer, freeze, and fail commands for system filters 372
43.6 Adding and removing headers in a system filter 373
43.7 Setting an errors address in a system filter 373
43.8 Per-address filtering 374

44. Message processing 375

44.1 Submission mode for non-local messages 375

44.2 Line endings ... 376
44.3 Unqualified addresses ... 376
44.4 The UUCP From line ... 377
44.5 Resent- header lines ... 377
44.6 The Auto-Submitted: header line ... 378
44.7 The Bcc: header line ... 378
44.8 The Date: header line ... 378
44.9 The Delivery-date: header line ... 378
44.10 The Envelope-to: header line ... 378
44.11 The From: header line ... 378
44.12 The Message-ID: header line ... 379
44.13 The Received: header line ... 379
44.14 The References: header line ... 379
44.15 The Return-path: header line ... 379
44.16 The Sender: header line ... 379
44.17 Adding and removing header lines in routers and transports ... 380
44.18 Constructed addresses ... 381
44.19 Case of local parts ... 381
44.20 Dots in local parts ... 382
44.21 Rewriting addresses ... 382

45. SMTP processing ... 383

45.1 Outgoing SMTP and LMTP over TCP/IP ... 383
45.2 Errors in outgoing SMTP ... 384
45.3 Incoming SMTP messages over TCP/IP ... 385
45.4 Unrecognized SMTP commands ... 387
45.5 Syntax and protocol errors in SMTP commands ... 387
45.6 Use of non-mail SMTP commands ... 387
45.7 The VRFY and EXPN commands ... 387
45.8 The ETRN command ... 387
45.9 Incoming local SMTP ... 388
45.10 Outgoing batched SMTP ... 388
45.11 Incoming batched SMTP ... 389

46. Customizing bounce and warning messages ... 390

46.1 Customizing bounce messages ... 390
46.2 Customizing warning messages ... 391

47. Some common configuration settings ... 392

47.1 Sending mail to a smart host ... 392
47.2 Using Exim to handle mailing lists ... 392
47.3 Syntax errors in mailing lists ... 392
47.4 Re-expansion of mailing lists ... 393
47.5 Closed mailing lists ... 393
47.6 Variable Envelope Return Paths (VERP) ... 394
47.7 Virtual domains ... 395
47.8 Multiple user mailboxes ... 396
47.9 Simplified vacation processing ... 397
47.10 Taking copies of mail ... 397
47.11 Intermittently connected hosts ... 397
47.12 Exim on the upstream server host ... 397
47.13 Exim on the intermittently connected client host ... 398

48. Using Exim as a non-queueing client ... 399

49. Log files **401**

49.1 Where the logs are written 401
49.2 Logging to local files that are periodically “cycled” 402
49.3 Datestamped log files 402
49.4 Logging to syslog 403
49.5 Log line flags 404
49.6 Logging message reception 404
49.7 Logging deliveries 405
49.8 Discarded deliveries 406
49.9 Deferred deliveries 406
49.10 Delivery failures 406
49.11 Fake deliveries 406
49.12 Completion 406
49.13 Summary of Fields in Log Lines 407
49.14 Other log entries 407
49.15 Reducing or increasing what is logged 408
49.16 Message log 411

50. Exim utilities **413**

50.1 Finding out what Exim processes are doing (exiwhat) 413
50.2 Selective queue listing (exiqgrep) 413
50.3 Summarizing the queue (exiqsumm) 414
50.4 Extracting specific information from the log (exigrep) 415
50.5 Selecting messages by various criteria (exipick) 415
50.6 Cycling log files (exicyclog) 415
50.7 Mail statistics (eximstats) 416
50.8 Checking access policy (exim_checkaccess) 417
50.9 Making DBM files (exim_dbmbuild) 417
50.10 Finding individual retry times (exinext) 418
50.11 Hints database maintenance 418
50.12 exim_dumpdb 418
50.13 exim_tidydb 419
50.14 exim_fixdb 419
50.15 Mailbox maintenance (exim_lock) 420

51. The Exim monitor **422**

51.1 Running the monitor 422
51.2 The stripcharts 422
51.3 Main action buttons 423
51.4 The log display 423
51.5 The queue display 424
51.6 The queue menu 424

52. Security considerations **426**

52.1 Building a more “hardened” Exim 426
52.2 Root privilege 426
52.3 Running Exim without privilege 427
52.4 Delivering to local files 428
52.5 IPv4 source routing 429
52.6 The VRFY, EXPN, and ETRN commands in SMTP 429
52.7 Privileged users 429
52.8 Spool files 429
52.9 Use of argv[0] 429
52.10 Use of %f formatting 429

52.11 Embedded Exim path 430
52.12 Use of sprintf() 430
52.13 Use of debug_printf() and log_write() 430
52.14 Use of strcat() and strcpy() 430

53. Format of spool files **431**

53.1 Format of the -H file 431

54. Adding new drivers or lookup types **436**

Options index **437**

Variables index **443**

Concept index **445**

1. Introduction

Exim is a mail transfer agent (MTA) for hosts that are running Unix or Unix-like operating systems. It was designed on the assumption that it would be run on hosts that are permanently connected to the Internet. However, it can be used on intermittently connected hosts with suitable configuration adjustments.

Configuration files currently exist for the following operating systems: AIX, BSD/OS (aka BSDI), Darwin (Mac OS X), DGUX, Dragonfly, FreeBSD, GNU/Hurd, GNU/Linux, HI-OSF (Hitachi), HI-UX, HP-UX, IRIX, MIPS RISCOS, NetBSD, OpenBSD, OpenUNIX, QNX, SCO, SCO SVR4.2 (aka UNIX-SV), Solaris (aka SunOS5), SunOS4, Tru64-Unix (formerly Digital UNIX, formerly DEC-OSF1), Ultrix, and Unixware. Some of these operating systems are no longer current and cannot easily be tested, so the configuration files may no longer work in practice.

There are also configuration files for compiling Exim in the Cygwin environment that can be installed on systems running Windows. However, this document does not contain any information about running Exim in the Cygwin environment.

The terms and conditions for the use and distribution of Exim are contained in the file *NOTICE*. Exim is distributed under the terms of the GNU General Public Licence, a copy of which may be found in the file *LICENCE*.

The use, supply or promotion of Exim for the purpose of sending bulk, unsolicited electronic mail is incompatible with the basic aims of the program, which revolve around the free provision of a service that enhances the quality of personal communications. The author of Exim regards indiscriminate mass-mailing as an antisocial, irresponsible abuse of the Internet.

Exim owes a great deal to Smail 3 and its author, Ron Karr. Without the experience of running and working on the Smail 3 code, I could never have contemplated starting to write a new MTA. Many of the ideas and user interfaces were originally taken from Smail 3, though the actual code of Exim is entirely new, and has developed far beyond the initial concept.

Many people, both in Cambridge and around the world, have contributed to the development and the testing of Exim, and to porting it to various operating systems. I am grateful to them all. The distribution now contains a file called *ACKNOWLEDGMENTS*, in which I have started recording the names of contributors.

1.1 Exim documentation

This edition of the Exim specification applies to version 4.68 of Exim. Substantive changes from the 4.67 edition are marked in some renditions of the document; this paragraph is so marked if the rendition is capable of showing a change indicator.

This document is very much a reference manual; it is not a tutorial. The reader is expected to have some familiarity with the SMTP mail transfer protocol and with general Unix system administration. Although there are some discussions and examples in places, the information is mostly organized in a way that makes it easy to look up, rather than in a natural order for sequential reading. Furthermore, the manual aims to cover every aspect of Exim in detail, including a number of rarely-used, special-purpose features that are unlikely to be of very wide interest.

An "easier" discussion of Exim which provides more in-depth explanatory, introductory, and tutorial material can be found in a book entitled *The Exim SMTP Mail Server* (second edition, 2007), published by UIT Cambridge (**http://www.uit.co.uk/exim-book/**).

This book also contains a chapter that gives a general introduction to SMTP and Internet mail. Inevitably, however, the book is unlikely to be fully up-to-date with the latest release of Exim. (Note that the earlier book about Exim, published by O'Reilly, covers Exim 3, and many things have changed in Exim 4.)

If you are using a Debian distribution of Exim, you will find information about Debian-specific features in the file */usr/share/doc/exim4-base/README.Debian*. The command *man update-exim.conf* is another source of Debian-specific information.

As the program develops, there may be features in newer versions that have not yet made it into this document, which is updated only when the most significant digit of the fractional part of the version number changes. Specifications of new features that are not yet in this manual are placed in the file *doc/NewStuff* in the Exim distribution.

Some features may be classified as "experimental". These may change incompatibly while they are developing, or even be withdrawn. For this reason, they are not documented in this manual. Information about experimental features can be found in the file *doc/experimental.txt*.

All changes to the program (whether new features, bug fixes, or other kinds of change) are noted briefly in the file called *doc/ChangeLog*.

This specification itself is available as an ASCII file in *doc/spec.txt* so that it can easily be searched with a text editor. Other files in the *doc* directory are:

OptionLists.txt	list of all options in alphabetical order
dbm.discuss.txt	discussion about DBM libraries
exim.8	a man page of Exim's command line options
experimental.txt	documentation of experimental features
filter.txt	specification of the filter language
pcrepattern.txt	specification of PCRE regular expressions
pcretest.txt	specification of the PCRE testing program
Exim3.upgrade	upgrade notes from release 2 to release 3
Exim4.upgrade	upgrade notes from release 3 to release 4

The main specification and the specification of the filtering language are also available in other formats (HTML, PostScript, PDF, and Texinfo). Section 1.6 below tells you how to get hold of these.

1.2 FTP and web sites

The primary site for Exim source distributions is currently the University of Cambridge's FTP site, whose contents are described in *Where to find the Exim distribution* below. In addition, there is a web site and an FTP site at **exim.org**. These are now also hosted at the University of Cambridge. The **exim.org** site was previously hosted for a number of years by Energis Squared, formerly Planet Online Ltd, whose support I gratefully acknowledge.

As well as Exim distribution tar files, the Exim web site contains a number of differently formatted versions of the documentation. A recent addition to the online information is the Exim wiki (**http://wiki.exim.org**), which contains what used to be a separate FAQ, as well as various other examples, tips, and know-how that have been contributed by Exim users.

An Exim Bugzilla exists at **http://bugs.exim.org**. You can use this to report bugs, and also to add items to the wish list. Please search first to check that you are not duplicating a previous entry.

1.3 Mailing lists

The following Exim mailing lists exist:

exim-users@exim.org	General discussion list
exim-dev@exim.org	Discussion of bugs, enhancements, etc.
exim-announce@exim.org	Moderated, low volume announcements list
exim-future@exim.org	Discussion of long-term development

You can subscribe to these lists, change your existing subscriptions, and view or search the archives via the mailing lists link on the Exim home page. If you are using a Debian distribution of Exim, you may wish to subscribe to the Debian-specific mailing list *pkg-exim4-users@lists.alioth.debian.org* via this web page:

http://lists.alioth.debian.org/mailman/listinfo/pkg-exim4-users

Please ask Debian-specific questions on this list and not on the general Exim lists.

1.4 Exim training

Training courses in Cambridge (UK) used to be run annually by the author of Exim, before he retired. At the time of writing, there are no plans to run further Exim courses in Cambridge. However, if that changes, relevant information will be posted at **http://www-tus.csx.cam.ac.uk/courses/exim/**.

1.5 Bug reports

Reports of obvious bugs can be emailed to *bugs@exim.org* or reported via the Bugzilla (**http://bugs.exim.org**). However, if you are unsure whether some behaviour is a bug or not, the best thing to do is to post a message to the *exim-dev* mailing list and have it discussed.

1.6 Where to find the Exim distribution

The master ftp site for the Exim distribution is

ftp://ftp.csx.cam.ac.uk/pub/software/email/exim

This is mirrored by

ftp://ftp.exim.org/pub/exim

The file references that follow are relative to the *exim* directories at these sites. There are now quite a number of independent mirror sites around the world. Those that I know about are listed in the file called *Mirrors*.

Within the *exim* directory there are subdirectories called *exim3* (for previous Exim 3 distributions), *exim4* (for the latest Exim 4 distributions), and *Testing* for testing versions. In the *exim4* subdirectory, the current release can always be found in files called

exim-n.nn.tar.gz
exim-n.nn.tar.bz2

where *n.nn* is the highest such version number in the directory. The two files contain identical data; the only difference is the type of compression. The *.bz2* file is usually a lot smaller than the *.gz* file.

The distributions are currently signed with Philip Hazel's GPG key. The corresponding public key is available from a number of keyservers, and there is also a copy in the file *Public-Key*. The signatures for the tar bundles are in:

exim-n.nn.tar.gz.sig
exim-n.nn.tar.bz2.sig

For each released version, the log of changes is made separately available in a separate file in the directory *ChangeLogs* so that it is possible to find out what has changed without having to download the entire distribution.

The main distribution contains ASCII versions of this specification and other documentation; other formats of the documents are available in separate files inside the *exim4* directory of the FTP site:

exim-html-n.nn.tar.gz
exim-pdf-n.nn.tar.gz
exim-postscript-n.nn.tar.gz
exim-texinfo-n.nn.tar.gz

These tar files contain only the *doc* directory, not the complete distribution, and are also available in *.bz2* as well as *.gz* forms.

1.7 Limitations

- Exim is designed for use as an Internet MTA, and therefore handles addresses in RFC 2822 domain format only. It cannot handle UUCP "bang paths", though simple two-component bang paths can be converted by a straightforward rewriting configuration. This restriction does not prevent Exim from being interfaced to UUCP as a transport mechanism, provided that domain addresses are used.

- Exim insists that every address it handles has a domain attached. For incoming local messages, domainless addresses are automatically qualified with a configured domain value. Configuration options specify from which remote systems unqualified addresses are acceptable. These are then qualified on arrival.
- The only external transport mechanisms that are currently implemented are SMTP and LMTP over a TCP/IP network (including support for IPv6). However, a pipe transport is available, and there are facilities for writing messages to files and pipes, optionally in *batched SMTP* format; these facilities can be used to send messages to other transport mechanisms such as UUCP, provided they can handle domain-style addresses. Batched SMTP input is also catered for.
- Exim is not designed for storing mail for dial-in 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-in hosts by other means.
- Although Exim does have basic facilities for scanning incoming messages, these are not comprehensive enough to do full virus or spam scanning. Such operations are best carried out using additional specialized software packages. If you compile Exim with the content-scanning extension, straightforward interfaces to a number of common scanners are provided.

1.8 Run time configuration

Exim's run time configuration is held in a single text file that is divided into a number of sections. The entries in this file consist of keywords and values, in the style of Smail 3 configuration files. A default configuration file which is suitable for simple online installations is provided in the distribution, and is described in chapter 7 below.

1.9 Calling interface

Like many MTAs, Exim has adopted the Sendmail command line interface so that it can be a straight replacement for */usr/lib/sendmail* or */usr/sbin/sendmail* when sending mail, but you do not need to know anything about Sendmail in order to run Exim. For actions other than sending messages, Sendmail-compatible options also exist, but those that produce output (for example, **-bp**, which lists the messages on the queue) do so in Exim's own format. There are also some additional options that are compatible with Smail 3, and some further options that are new to Exim. Chapter 5 documents all Exim's command line options. This information is automatically made into the man page that forms part of the Exim distribution.

Control of messages on the queue can be done via certain privileged command line options. There is also an optional monitor program called *eximon*, which displays current information in an X window, and which contains a menu interface to Exim's command line administration options.

1.10 Terminology

The *body* of a message is the actual data that the sender wants to transmit. It is the last part of a message, and is separated from the *header* (see below) by a blank line.

When a message cannot be delivered, it is normally returned to the sender in a delivery failure message or a "non-delivery report" (NDR). The term *bounce* is commonly used for this action, and the error reports are often called *bounce messages*. This is a convenient shorthand for "delivery failure error report". Such messages have an empty sender address in the message's *envelope* (see below) to ensure that they cannot themselves give rise to further bounce messages.

The term *default* appears frequently in this manual. It is used to qualify a value which is used in the absence of any setting in the configuration. It may also qualify an action which is taken unless a configuration setting specifies otherwise.

The term *defer* is used when the delivery of a message to a specific destination cannot immediately take place for some reason (a remote host may be down, or a user's local mailbox may be full). Such deliveries are *deferred* until a later time.

The word *domain* is sometimes used to mean all but the first component of a host's name. It is *not* used in that sense here, where it normally refers to the part of an email address following the @ sign.

A message in transit has an associated *envelope*, as well as a header and a body. The envelope contains a sender address (to which bounce messages should be delivered), and any number of recipient addresses. References to the sender or the recipients of a message usually mean the addresses in the envelope. An MTA uses these addresses for delivery, and for returning bounce messages, not the addresses that appear in the header lines.

The *header* of a message is the first part of a message's text, consisting of a number of lines, each of which has a name such as *From:*, *To:*, *Subject:*, etc. Long header lines can be split over several text lines by indenting the continuations. The header is separated from the body by a blank line.

The term *local part*, which is taken from RFC 2822, is used to refer to that part of an email address that precedes the @ sign. The part that follows the @ sign is called the *domain* or *mail domain*.

The terms *local delivery* and *remote delivery* are used to distinguish delivery to a file or a pipe on the local host from delivery by SMTP over TCP/IP to another host. As far as Exim is concerned, all hosts other than the host it is running on are *remote*.

Return path is another name that is used for the sender address in a message's envelope.

The term *queue* is used to refer to the set of messages awaiting delivery, because this term is in widespread use in the context of MTAs. However, in Exim's case the reality is more like a pool than a queue, because there is normally no ordering of waiting messages.

The term *queue runner* is used to describe a process that scans the queue and attempts to deliver those messages whose retry times have come. This term is used by other MTAs, and also relates to the command **runq**, but in Exim the waiting messages are normally processed in an unpredictable order.

The term *spool directory* is used for a directory in which Exim keeps the messages on its queue – that is, those that it is in the process of delivering. This should not be confused with the directory in which local mailboxes are stored, which is called a "spool directory" by some people. In the Exim documentation, "spool" is always used in the first sense.

2. Incorporated code

A number of pieces of external code are included in the Exim distribution.

- Regular expressions are supported in the main Exim program and in the Exim monitor using the freely-distributable PCRE library, copyright © University of Cambridge. The source is distributed in the directory *src/pcre*. However, this is a cut-down version of PCRE. If you want to use the PCRE library in other programs, you should obtain and install the full version of the library from **ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre**.
- Support for the cdb (Constant DataBase) lookup method is provided by code contributed by Nigel Metheringham of (at the time he contributed it) Planet Online Ltd. The implementation is completely contained within the code of Exim. It does not link against an external cdb library. The code contains the following statements:

  Copyright © 1998 Nigel Metheringham, Planet Online Ltd

  This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. This code implements Dan Bernstein's Constant DataBase (cdb) spec. Information, the spec and sample code for cdb can be obtained from **http://www.pobox.com/~djb/cdb.html**. This implementation borrows some code from Dan Bernstein's implementation (which has no license restrictions applied to it).
- Client support for Microsoft's *Secure Password Authentication* is provided by code contributed by Marc Prud'hommeaux. Server support was contributed by Tom Kistner. This includes code taken from the Samba project, which is released under the Gnu GPL.
- Support for calling the Cyrus *pwcheck* and *saslauthd* daemons is provided by code taken from the Cyrus-SASL library and adapted by Alexander S. Sabourenkov. The permission notice appears below, in accordance with the conditions expressed therein.

  Copyright © 2001 Carnegie Mellon University. All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  (1) Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

  (2) Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

  (3) The name "Carnegie Mellon University" must not be used to endorse or promote products derived from this software without prior written permission. For permission or any other legal details, please contact

  Office of Technology Transfer
  Carnegie Mellon University
  5000 Forbes Avenue
  Pittsburgh, PA 15213-3890
  (412) 268-4387, fax: (412) 268-7395
  tech-transfer@andrew.cmu.edu

  (4) Redistributions of any form whatsoever must retain the following acknowledgment:

  "This product includes software developed by Computing Services at Carnegie Mellon University (**http://www.cmu.edu/computing/**."

  CARNEGIE MELLON UNIVERSITY DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES

OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL CARNEGIE MELLON UNIVERSITY BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

- The Exim Monitor program, which is an X-Window application, includes modified versions of the Athena StripChart and TextPop widgets. This code is copyright by DEC and MIT, and their permission notice appears below, in accordance with the conditions expressed therein.

  Copyright 1987, 1988 by Digital Equipment Corporation, Maynard, Massachusetts, and the Massachusetts Institute of Technology, Cambridge, Massachusetts.

  All Rights Reserved

  Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the names of Digital or MIT not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

  DIGITAL DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL DIGITAL BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

- Many people have contributed code fragments, some large, some small, that were not covered by any specific licence requirements. It is assumed that the contributors are happy to see their code incorporated into Exim under the GPL.

3. How Exim receives and delivers mail

3.1 Overall philosophy

Exim is designed to work efficiently on systems that are permanently connected to the Internet and are handling a general mix of mail. In such circumstances, most messages can be delivered immediately. Consequently, Exim does not maintain independent queues of messages for specific domains or hosts, though it does try to send several messages in a single SMTP connection after a host has been down, and it also maintains per-host retry information.

3.2 Policy control

Policy controls are now an important feature of MTAs that are connected to the Internet. Perhaps their most important job is to stop MTAs being abused as "open relays" by misguided individuals who send out vast amounts of unsolicited junk, and want to disguise its source. Exim provides flexible facilities for specifying policy controls on incoming mail:

- Exim 4 (unlike previous versions of Exim) implements policy controls on incoming mail by means of *Access Control Lists* (ACLs). Each list is a series of statements that may either grant or deny access. ACLs can be used at several places in the SMTP dialogue while receiving a message from a remote host. However, the most common places are after each RCPT command, and at the very end of the message. The sysadmin can specify conditions for accepting or rejecting individual recipients or the entire message, respectively, at these two points (see chapter 40). Denial of access results in an SMTP error code.

- An ACL is also available for locally generated, non-SMTP messages. In this case, the only available actions are to accept or deny the entire message.

- When Exim is compiled with the content-scanning extension, facilities are provided in the ACL mechanism for passing the message to external virus and/or spam scanning software. The result of such a scan is passed back to the ACL, which can then use it to decide what to do with the message.

- When a message has been received, either from a remote host or from the local host, but before the final acknowledgment has been sent, a locally supplied C function called *local_scan()* can be run to inspect the message and decide whether to accept it or not (see chapter 42). If the message is accepted, the list of recipients can be modified by the function.

- Using the *local_scan()* mechanism is another way of calling external scanner software. The **SA-Exim** add-on package works this way. It does not require Exim to be compiled with the content-scanning extension.

- After a message has been accepted, a further checking mechanism is available in the form of the *system filter* (see chapter 43). This runs at the start of every delivery process.

3.3 User filters

In a conventional Exim configuration, users are able to run private filters by setting up appropriate *.forward* files in their home directories. See chapter 22 (about the *redirect* router) for the configuration needed to support this, and the separate document entitled *Exim's interfaces to mail filtering* for user details. Two different kinds of filtering are available:

- Sieve filters are written in the standard filtering language that is defined by RFC 3028.

- Exim filters are written in a syntax that is unique to Exim, but which is more powerful than Sieve, which it pre-dates.

User filters are run as part of the routing process, described below.

3.4 Message identification

Every message handled by Exim is given a *message id* which is sixteen characters long. It is divided into three parts, separated by hyphens, for example `16VDhn-0001bo-D3`. Each part is a sequence of letters and digits, normally encoding numbers in base 62. However, in the Darwin operating system (Mac OS X) and when Exim is compiled to run under Cygwin, base 36 (avoiding the use of lower case letters) is used instead, because the message id is used to construct file names, and the names of files in those systems are not always case-sensitive.

The detail of the contents of the message id have changed as Exim has evolved. Earlier versions relied on the operating system not re-using a process id (pid) within one second. On modern operating systems, this assumption can no longer be made, so the algorithm had to be changed. To retain backward compatibility, the format of the message id was retained, which is why the following rules are somewhat eccentric:

- The first six characters of the message id are the time at which the message started to be received, to a granularity of one second. That is, this field contains the number of seconds since the start of the epoch (the normal Unix way of representing the date and time of day).
- After the first hyphen, the next six characters are the id of the process that received the message.
- There are two different possibilities for the final two characters:

  (1) If **localhost_number** is not set, this value is the fractional part of the time of reception, normally in units of 1/2000 of a second, but for systems that must use base 36 instead of base 62 (because of case-insensitive file systems), the units are 1/1000 of a second.

  (2) If **localhost_number** is set, it is multiplied by 200 (100) and added to the fractional part of the time, which in this case is in units of 1/200 (1/100) of a second.

After a message has been received, Exim waits for the clock to tick at the appropriate resolution before proceeding, so that if another message is received by the same process, or by another process with the same (re-used) pid, it is guaranteed that the time will be different. In most cases, the clock will already have ticked while the message was being received.

3.5 Receiving mail

The only way Exim can receive mail from another host is using SMTP over TCP/IP, in which case the sender and recipient addresses are transferred using SMTP commands. However, from a locally running process (such as a user's MUA), there are several possibilities:

- If the process runs Exim with the **-bm** option, the message is read non-interactively (usually via a pipe), with the recipients taken from the command line, or from the body of the message if **-t** is also used.
- If the process runs Exim with the **-bS** option, the message is also read non-interactively, but in this case the recipients are listed at the start of the message in a series of SMTP RCPT commands, terminated by a DATA command. This is so-called "batch SMTP" format, but it isn't really SMTP. The SMTP commands are just another way of passing envelope addresses in a non-interactive submission.
- If the process runs Exim with the **-bs** option, the message is read interactively, using the SMTP protocol. A two-way pipe is normally used for passing data between the local process and the Exim process. This is "real" SMTP and is handled in the same way as SMTP over TCP/IP. For example, the ACLs for SMTP commands are used for this form of submission.
- A local process may also make a TCP/IP call to the host's loopback address (127.0.0.1) or any other of its IP addresses. When receiving messages, Exim does not treat the loopback address specially. It treats all such connections in the same way as connections from other hosts.

In the three cases that do not involve TCP/IP, the sender address is constructed from the login name of the user that called Exim and a default qualification domain (which can be set by the **qualify_domain** configuration option). For local or batch SMTP, a sender address that is passed using the SMTP MAIL command is ignored. However, the system administrator may allow certain users ("trusted

users") to specify a different sender address unconditionally, or all users to specify certain forms of different sender address. The **-f** option or the SMTP MAIL command is used to specify these different addresses. See section 5.2 for details of trusted users, and the **untrusted_set_sender** option for a way of allowing untrusted users to change sender addresses.

Messages received by either of the non-interactive mechanisms are subject to checking by the non-SMTP ACL, if one is defined. Messages received using SMTP (either over TCP/IP, or interacting with a local process) can be checked by a number of ACLs that operate at different times during the SMTP session. Either individual recipients, or the entire message, can be rejected if local policy requirements are not met. The *local_scan()* function (see chapter 42) is run for all incoming messages.

Exim can be configured not to start a delivery process when a message is received; this can be unconditional, or depend on the number of incoming SMTP connections or the system load. In these situations, new messages wait on the queue until a queue runner process picks them up. However, in standard configurations under normal conditions, delivery is started as soon as a message is received.

3.6 Handling an incoming message

When Exim accepts a message, it writes two files in its spool directory. The first contains the envelope information, the current status of the message, and the header lines, and the second contains the body of the message. The names of the two spool files consist of the message id, followed by `-H` for the file containing the envelope and header, and `-D` for the data file.

By default all these message files are held in a single directory called *input* inside the general Exim spool directory. Some operating systems do not perform very well if the number of files in a directory gets large; to improve performance in such cases, the **split_spool_directory** option can be used. This causes Exim to split up the input files into 62 sub-directories whose names are single letters or digits. When this is done, the queue is processed one sub-directory at a time instead of all at once, which can improve overall performance even when there are not enough files in each directory to affect file system performance.

The envelope information consists of the address of the message's sender and the addresses of the recipients. This information is entirely separate from any addresses contained in the header lines. The status of the message includes a list of recipients who have already received the message. The format of the first spool file is described in chapter 53.

Address rewriting that is specified in the rewrite section of the configuration (see chapter 31) is done once and for all on incoming addresses, both in the header lines and the envelope, at the time the message is accepted. If during the course of delivery additional addresses are generated (for example, via aliasing), these new addresses are rewritten as soon as they are generated. At the time a message is actually delivered (transported) further rewriting can take place; because this is a transport option, it can be different for different forms of delivery. It is also possible to specify the addition or removal of certain header lines at the time the message is delivered (see chapters 15 and 24).

3.7 Life of a message

A message remains in the spool directory until it is completely delivered to its recipients or to an error address, or until it is deleted by an administrator or by the user who originally created it. In cases when delivery cannot proceed – for example, when a message can neither be delivered to its recipients nor returned to its sender, the message is marked "frozen" on the spool, and no more deliveries are attempted.

An administrator can "thaw" such messages when the problem has been corrected, and can also freeze individual messages by hand if necessary. In addition, an administrator can force a delivery error, causing a bounce message to be sent.

There are options called **ignore_bounce_errors_after** and **timeout_frozen_after**, which discard frozen messages after a certain time. The first applies only to frozen bounces, the second to any frozen messages.

While Exim is working on a message, it writes information about each delivery attempt to its main log file. This includes successful, unsuccessful, and delayed deliveries for each recipient (see chapter

49). The log lines are also written to a separate *message log* file for each message. These logs are solely for the benefit of the administrator, and are normally deleted along with the spool files when processing of a message is complete. The use of individual message logs can be disabled by setting **no_message_logs**; this might give an improvement in performance on very busy systems.

All the information Exim itself needs to set up a delivery is kept in the first spool file, along with the header lines. When a successful delivery occurs, the address is immediately written at the end of a journal file, whose name is the message id followed by `-J`. At the end of a delivery run, if there are some addresses left to be tried again later, the first spool file (the `-H` file) is updated to indicate which these are, and the journal file is then deleted. Updating the spool file is done by writing a new file and renaming it, to minimize the possibility of data loss.

Should the system or the program crash after a successful delivery but before the spool file has been updated, the journal is left lying around. The next time Exim attempts to deliver the message, it reads the journal file and updates the spool file before proceeding. This minimizes the chances of double deliveries caused by crashes.

3.8 Processing an address for delivery

The main delivery processing elements of Exim are called *routers* and *transports*, and collectively these are known as *drivers*. Code for a number of them is provided in the source distribution, and compile-time options specify which ones are included in the binary. Run time options specify which ones are actually used for delivering messages.

Each driver that is specified in the run time configuration is an *instance* of that particular driver type. Multiple instances are allowed; for example, you can set up several different *smtp* transports, each with different option values that might specify different ports or different timeouts. Each instance has its own identifying name. In what follows we will normally use the instance name when discussing one particular instance (that is, one specific configuration of the driver), and the generic driver name when discussing the driver's features in general.

A *router* is a driver that operates on an address, either determining how its delivery should happen, by assigning it to a specific transport, or converting the address into one or more new addresses (for example, via an alias file). A router may also explicitly choose to fail an address, causing it to be bounced.

A *transport* is a driver that transmits a copy of the message from Exim's spool to some destination. There are two kinds of transport: for a *local* transport, the destination is a file or a pipe on the local host, whereas for a *remote* transport the destination is some other host. A message is passed to a specific transport as a result of successful routing. If a message has several recipients, it may be passed to a number of different transports.

An address is processed by passing it to each configured router instance in turn, subject to certain preconditions, until a router accepts the address or specifies that it should be bounced. We will describe this process in more detail shortly. First, as a simple example, we consider how each recipient address in a message is processed in a small configuration of three routers.

To make this a more concrete example, it is described in terms of some actual routers, but remember, this is only an example. You can configure Exim's routers in many different ways, and there may be any number of routers in a configuration.

The first router that is specified in a configuration is often one that handles addresses in domains that are not recognized specially by the local host. These are typically addresses for arbitrary domains on the Internet. A precondition is set up which looks for the special domains known to the host (for example, its own domain name), and the router is run for addresses that do *not* match. Typically, this is a router that looks up domains in the DNS in order to find the hosts to which this address routes. If it succeeds, the address is assigned to a suitable SMTP transport; if it does not succeed, the router is configured to fail the address.

The second router is reached only when the domain is recognized as one that "belongs" to the local host. This router does redirection – also known as aliasing and forwarding. When it generates one or more new addresses from the original, each of them is routed independently from the start. Otherwise,

the router may cause an address to fail, or it may simply decline to handle the address, in which case the address is passed to the next router.

The final router in many configurations is one that checks to see if the address belongs to a local mailbox. The precondition may involve a check to see if the local part is the name of a login account, or it may look up the local part in a file or a database. If its preconditions are not met, or if the router declines, we have reached the end of the routers. When this happens, the address is bounced.

3.9 Processing an address for verification

As well as being used to decide how to deliver to an address, Exim's routers are also used for *address verification*. Verification can be requested as one of the checks to be performed in an ACL for incoming messages, on both sender and recipient addresses, and it can be tested using the **-bv** and **-bvs** command line options.

When an address is being verified, the routers are run in "verify mode". This does not affect the way the routers work, but it is a state that can be detected. By this means, a router can be skipped or made to behave differently when verifying. A common example is a configuration in which the first router sends all messages to a message-scanning program, unless they have been previously scanned. Thus, the first router accepts all addresses without any checking, making it useless for verifying. Normally, the **no_verify** option would be set for such a router, causing it to be skipped in verify mode.

3.10 Running an individual router

As explained in the example above, a number of preconditions are checked before running a router. If any are not met, the router is skipped, and the address is passed to the next router. When all the preconditions on a router *are* met, the router is run. What happens next depends on the outcome, which is one of the following:

- *accept*: The router accepts the address, and either assigns it to a transport, or generates one or more "child" addresses. Processing the original address ceases, unless the **unseen** option is set on the router. This option can be used to set up multiple deliveries with different routing (for example, for keeping archive copies of messages). When **unseen** is set, the address is passed to the next router. Normally, however, an *accept* return marks the end of routing.

  Any child addresses generated by the router are processed independently, starting with the first router by default. It is possible to change this by setting the **redirect_router** option to specify which router to start at for child addresses. Unlike **pass_router** (see below) the router specified by **redirect_router** may be anywhere in the router configuration.

- *pass*: The router recognizes the address, but cannot handle it itself. It requests that the address be passed to another router. By default the address is passed to the next router, but this can be changed by setting the **pass_router** option. However, (unlike **redirect_router**) the named router must be below the current router (to avoid loops).

- *decline*: The router declines to accept the address because it does not recognize it at all. By default, the address is passed to the next router, but this can be prevented by setting the **no_more** option. When **no_more** is set, all the remaining routers are skipped. In effect, **no_more** converts *decline* into *fail*.

- *fail*: The router determines that the address should fail, and queues it for the generation of a bounce message. There is no further processing of the original address unless **unseen** is set on the router.

- *defer*: The router cannot handle the address at the present time. (A database may be offline, or a DNS lookup may have timed out.) No further processing of the address happens in this delivery attempt. It is tried again next time the message is considered for delivery.

- *error*: There is some error in the router (for example, a syntax error in its configuration). The action is as for defer.

If an address reaches the end of the routers without having been accepted by any of them, it is bounced as unrouteable. The default error message in this situation is "unrouteable address", but you

can set your own message by making use of the **cannot_route_message** option. This can be set for any router; the value from the last router that "saw" the address is used.

Sometimes while routing you want to fail a delivery when some conditions are met but others are not, instead of passing the address on for further routing. You can do this by having a second router that explicitly fails the delivery when the relevant conditions are met. The *redirect* router has a "fail" facility for this purpose.

3.11 Duplicate addresses

Once routing is complete, Exim scans the addresses that are assigned to local and remote transports, and discards any duplicates that it finds. During this check, local parts are treated as case-sensitive. This happens only when actually delivering a message; when testing routers with **-bt**, all the routed addresses are shown.

3.12 Router preconditions

The preconditions that are tested for each router are listed below, in the order in which they are tested. The individual configuration options are described in more detail in chapter 15.

- The **local_part_prefix** and **local_part_suffix** options can specify that the local parts handled by the router may or must have certain prefixes and/or suffixes. If a mandatory affix (prefix or suffix) is not present, the router is skipped. These conditions are tested first. When an affix is present, it is removed from the local part before further processing, including the evaluation of any other conditions.
- Routers can be designated for use only when not verifying an address, that is, only when routing it for delivery (or testing its delivery routing). If the **verify** option is set false, the router is skipped when Exim is verifying an address. Setting the **verify** option actually sets two options, **verify_sender** and **verify_recipient**, which independently control the use of the router for sender and recipient verification. You can set these options directly if you want a router to be used for only one type of verification.
- If the **address_test** option is set false, the router is skipped when Exim is run with the **-bt** option to test an address routing. This can be helpful when the first router sends all new messages to a scanner of some sort; it makes it possible to use **-bt** to test subsequent delivery routing without having to simulate the effect of the scanner.
- Routers can be designated for use only when verifying an address, as opposed to routing it for delivery. The **verify_only** option controls this.
- Individual routers can be explicitly skipped when running the routers to check an address given in the SMTP EXPN command (see the **expn** option).
- If the **domains** option is set, the domain of the address must be in the set of domains that it defines.
- If the **local_parts** option is set, the local part of the address must be in the set of local parts that it defines. If **local_part_prefix** or **local_part_suffix** is in use, the prefix or suffix is removed from the local part before this check. If you want to do precondition tests on local parts that include affixes, you can do so by using a **condition** option (see below) that uses the variables *$local_part*, *$local_part_prefix*, and *$local_part_suffix* as necessary.
- If the **check_local_user** option is set, the local part must be the name of an account on the local host. If this check succeeds, the uid and gid of the local user are placed in *$local_user_uid* and *$local_user_gid* and the user's home directory is placed in *$home*; these values can be used in the remaining preconditions.
- If the **router_home_directory** option is set, it is expanded at this point, because it overrides the value of *$home*. If this expansion were left till later, the value of *$home* as set by **check_local_user** would be used in subsequent tests. Having two different values of *$home* in the same router could lead to confusion.

- If the **senders** option is set, the envelope sender address must be in the set of addresses that it defines.
- If the **require_files** option is set, the existence or non-existence of specified files is tested.
- If the **condition** option is set, it is evaluated and tested. This option uses an expanded string to allow you to set up your own custom preconditions. Expanded strings are described in chapter 11.

Note that **require_files** comes near the end of the list, so you cannot use it to check for the existence of a file in which to lookup up a domain, local part, or sender. However, as these options are all expanded, you can use the **exists** expansion condition to make such tests within each condition. The **require_files** option is intended for checking files that the router may be going to use internally, or which are needed by a specific transport (for example, *.procmailrc*).

3.13 Delivery in detail

When a message is to be delivered, the sequence of events is as follows:

- If a system-wide filter file is specified, the message is passed to it. The filter may add recipients to the message, replace the recipients, discard the message, cause a new message to be generated, or cause the message delivery to fail. The format of the system filter file is the same as for Exim user filter files, described in the separate document entitled *Exim's interfaces to mail filtering*. (**Note**: Sieve cannot be used for system filter files.)

  Some additional features are available in system filters – see chapter 43 for details. Note that a message is passed to the system filter only once per delivery attempt, however many recipients it has. However, if there are several delivery attempts because one or more addresses could not be immediately delivered, the system filter is run each time. The filter condition **first_delivery** can be used to detect the first run of the system filter.
- Each recipient address is offered to each configured router in turn, subject to its preconditions, until one is able to handle it. If no router can handle the address, that is, if they all decline, the address is failed. Because routers can be targeted at particular domains, several locally handled domains can be processed entirely independently of each other.
- A router that accepts an address may assign it to a local or a remote transport. However, the transport is not run at this time. Instead, the address is placed on a list for the particular transport, which will be run later. Alternatively, the router may generate one or more new addresses (typically from alias, forward, or filter files). New addresses are fed back into this process from the top, but in order to avoid loops, a router ignores any address which has an identically-named ancestor that was processed by itself.
- When all the routing has been done, addresses that have been successfully handled are passed to their assigned transports. When local transports are doing real local deliveries, they handle only one address at a time, but if a local transport is being used as a pseudo-remote transport (for example, to collect batched SMTP messages for transmission by some other means) multiple addresses can be handled. Remote transports can always handle more than one address at a time, but can be configured not to do so, or to restrict multiple addresses to the same domain.
- Each local delivery to a file or a pipe runs in a separate process under a non-privileged uid, and these deliveries are run one at a time. Remote deliveries also run in separate processes, normally under a uid that is private to Exim ("the Exim user"), but in this case, several remote deliveries can be run in parallel. The maximum number of simultaneous remote deliveries for any one message is set by the **remote_max_parallel** option. The order in which deliveries are done is not defined, except that all local deliveries happen before any remote deliveries.
- When it encounters a local delivery during a queue run, Exim checks its retry database to see if there has been a previous temporary delivery failure for the address before running the local transport. If there was a previous failure, Exim does not attempt a new delivery until the retry time for the address is reached. However, this happens only for delivery attempts that are part of a queue run. Local deliveries are always attempted when delivery immediately follows message reception, even if retry times are set for them. This makes for better behaviour if one particular message is causing problems (for example, causing quota overflow, or provoking an error in a filter file).

- Remote transports do their own retry handling, since an address may be deliverable to one of a number of hosts, each of which may have a different retry time. If there have been previous temporary failures and no host has reached its retry time, no delivery is attempted, whether in a queue run or not. See chapter 32 for details of retry strategies.
- If there were any permanent errors, a bounce message is returned to an appropriate address (the sender in the common case), with details of the error for each failing address. Exim can be configured to send copies of bounce messages to other addresses.
- If one or more addresses suffered a temporary failure, the message is left on the queue, to be tried again later. Delivery of these addresses is said to be *deferred*.
- When all the recipient addresses have either been delivered or bounced, handling of the message is complete. The spool files and message log are deleted, though the message log can optionally be preserved if required.

3.14 Retry mechanism

Exim's mechanism for retrying messages that fail to get delivered at the first attempt is the queue runner process. You must either run an Exim daemon that uses the **-q** option with a time interval to start queue runners at regular intervals, or use some other means (such as *cron*) to start them. If you do not arrange for queue runners to be run, messages that fail temporarily at the first attempt will remain on your queue for ever. A queue runner process works its way through the queue, one message at a time, trying each delivery that has passed its retry time. You can run several queue runners at once.

Exim uses a set of configured rules to determine when next to retry the failing address (see chapter 32). These rules also specify when Exim should give up trying to deliver to the address, at which point it generates a bounce message. If no retry rules are set for a particular host, address, and error combination, no retries are attempted, and temporary errors are treated as permanent.

3.15 Temporary delivery failure

There are many reasons why a message may not be immediately deliverable to a particular address. Failure to connect to a remote machine (because it, or the connection to it, is down) is one of the most common. Temporary failures may be detected during routing as well as during the transport stage of delivery. Local deliveries may be delayed if NFS files are unavailable, or if a mailbox is on a file system where the user is over quota. Exim can be configured to impose its own quotas on local mailboxes; where system quotas are set they will also apply.

If a host is unreachable for a period of time, a number of messages may be waiting for it by the time it recovers, and sending them in a single SMTP connection is clearly beneficial. Whenever a delivery to a remote host is deferred,

Exim makes a note in its hints database, and whenever a successful SMTP delivery has happened, it looks to see if any other messages are waiting for the same host. If any are found, they are sent over the same SMTP connection, subject to a configuration limit as to the maximum number in any one connection.

3.16 Permanent delivery failure

When a message cannot be delivered to some or all of its intended recipients, a bounce message is generated. Temporary delivery failures turn into permanent errors when their timeout expires. All the addresses that fail in a given delivery attempt are listed in a single message. If the original message has many recipients, it is possible for some addresses to fail in one delivery attempt and others to fail subsequently, giving rise to more than one bounce message. The wording of bounce messages can be customized by the administrator. See chapter 46 for details.

Bounce messages contain an *X-Failed-Recipients:* header line that lists the failed addresses, for the benefit of programs that try to analyse such messages automatically.

A bounce message is normally sent to the sender of the original message, as obtained from the message's envelope. For incoming SMTP messages, this is the address given in the MAIL command. However, when an address is expanded via a forward or alias file, an alternative address can be specified for delivery failures of the generated addresses. For a mailing list expansion (see section 47.2) it is common to direct bounce messages to the manager of the list.

3.17 Failures to deliver bounce messages

If a bounce message (either locally generated or received from a remote host) itself suffers a permanent delivery failure, the message is left on the queue, but it is frozen, awaiting the attention of an administrator. There are options that can be used to make Exim discard such failed messages, or to keep them for only a short time (see **timeout_frozen_after** and **ignore_bounce_errors_after**).

4. Building and installing Exim

4.1 Unpacking

Exim is distributed as a gzipped or bzipped tar file which, when unpacked, creates a directory with the name of the current release (for example, *exim-4.68*) into which the following files are placed:

ACKNOWLEDGMENTS	contains some acknowledgments
CHANGES	contains a reference to where changes are documented
LICENCE	the GNU General Public Licence
Makefile	top-level make file
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 are created:

Local	an empty directory for local configuration files
OS	OS-specific files
doc	documentation files
exim_monitor	source files for the Exim monitor
scripts	scripts used in the build process
src	remaining source files
util	independent utilities

The main utility programs are contained in the *src* directory, and are built with the Exim binary. The *util* directory contains a few optional scripts that may be useful to some sites.

4.2 Multiple machine architectures and operating systems

The building process for Exim is arranged to make it easy to build binaries for a number of different architectures and operating systems from the same set of source files. Compilation does not take place in the *src* directory. Instead, a *build directory* is created for each architecture and operating system. Symbolic links to the sources are installed in this directory, which is where the actual building takes place. In most cases, Exim can discover the machine architecture and operating system for itself, but the defaults can be overridden if necessary.

4.3 DBM libraries

Even if you do not use any DBM files in your configuration, Exim still needs a DBM library in order to operate, because it uses indexed files for its hints databases. Unfortunately, there are a number of DBM libraries in existence, and different operating systems often have different ones installed.

If you are using Solaris, IRIX, one of the modern BSD systems, or a modern Linux distribution, the DBM configuration should happen automatically, and you may be able to ignore this section. Otherwise, you may have to learn more than you would like about DBM libraries from what follows.

Licensed versions of Unix normally contain a library of DBM functions operating via the *ndbm* interface, and this is what Exim expects by default. Free versions of Unix seem to vary in what they contain as standard. In particular, some early versions of Linux have no default DBM library, and different distributors have chosen to bundle different libraries with their packaged versions. However, the more recent releases seem to have standardized on the Berkeley DB library.

Different DBM libraries have different conventions for naming the files they use. When a program opens a file called *dbmfile*, there are several possibilities:

(1) A traditional *ndbm* implementation, such as that supplied as part of Solaris, operates on two files called *dbmfile.dir* and *dbmfile.pag*.

(2) The GNU library, *gdbm*, operates on a single file. If used via its *ndbm* compatibility interface it makes two different hard links to it with names *dbmfile.dir* and *dbmfile.pag*, but if used via its native interface, the file name is used unmodified.

(3) The Berkeley DB package, if called via its *ndbm* compatibility interface, operates on a single file called *dbmfile.db*, but otherwise looks to the programmer exactly the same as the traditional *ndbm* implementation.

(4) If the Berkeley package is used in its native mode, it operates on a single file called *dbmfile*; the programmer's interface is somewhat different to the traditional *ndbm* interface.

(5) To complicate things further, there are several very different versions of the Berkeley DB package. Version 1.85 was stable for a very long time, releases 2.*x* and 3.*x* were current for a while, but the latest versions are now numbered 4.*x*. Maintenance of some of the earlier releases has ceased. All versions of Berkeley DB can be obtained from **http://www.sleepycat.com/**.

(6) Yet another DBM library, called *tdb*, is available from **http://download.sourceforge.net/tdb**. It has its own interface, and also operates on a single file.

Exim and its utilities can be compiled to use any of these interfaces. In order to use any version of the Berkeley DB package in native mode, you must set USE_DB in an appropriate configuration file (typically *Local/Makefile*). For example:

```
USE_DB=yes
```

Similarly, for gdbm you set USE_GDBM, and for tdb you set USE_TDB. An error is diagnosed if you set more than one of these.

At the lowest level, the build-time configuration sets none of these options, thereby assuming an interface of type (1). However, some operating system configuration files (for example, those for the BSD operating systems and Linux) assume type (4) by setting USE_DB as their default, and the configuration files for Cygwin set USE_GDBM. Anything you set in *Local/Makefile*, however, overrides these system defaults.

As well as setting USE_DB, USE_GDBM, or USE_TDB, it may also be necessary to set DBMLIB, to cause inclusion of the appropriate library, as in one of these lines:

```
DBMLIB = -ldb
DBMLIB = -ltdb
```

Settings like that will work if the DBM library is installed in the standard place. Sometimes it is not, and the library's header file may also not be in the default path. You may need to set INCLUDE to specify where the header file is, and to specify the path to the library more fully in DBMLIB, as in this example:

```
INCLUDE=-I/usr/local/include/db-4.1
DBMLIB=/usr/local/lib/db-4.1/libdb.a
```

There is further detailed discussion about the various DBM libraries in the file *doc/dbm.discuss.txt* in the Exim distribution.

4.4 Pre-building configuration

Before building Exim, a local configuration file that specifies options independent of any operating system has to be created with the name *Local/Makefile*. A template for this file is supplied as the file *src/EDITME*, and it contains full descriptions of all the option settings therein. These descriptions are therefore not repeated here. If you are building Exim for the first time, the simplest thing to do is to copy *src/EDITME* to *Local/Makefile*, then read it and edit it appropriately.

There are three settings that you must supply, because Exim will not build without them. They are the location of the run time configuration file (CONFIGURE_FILE), the directory in which Exim binaries will be installed (BIN_DIRECTORY), and the identity of the Exim user (EXIM_USER and maybe EXIM_GROUP as well). The value of CONFIGURE_FILE can in fact be a colon-separated list of file names; Exim uses the first of them that exists.

There are a few other parameters that can be specified either at build time or at run time, to enable the same binary to be used on a number of different machines. However, if the locations of Exim's spool directory and log file directory (if not within the spool directory) are fixed, it is recommended that

you specify them in *Local/Makefile* instead of at run time, so that errors detected early in Exim's execution (such as a malformed configuration file) can be logged.

Exim's interfaces for calling virus and spam scanning software directly from access control lists are not compiled by default. If you want to include these facilities, you need to set

```
WITH_CONTENT_SCAN=yes
```

in your *Local/Makefile*. For details of the facilities themselves, see chapter 41.

If you are going to build the Exim monitor, a similar configuration process is required. The file *exim_monitor/EDITME* must be edited appropriately for your installation and saved under the name *Local/eximon.conf*. If you are happy with the default settings described in *exim_monitor/EDITME*, *Local/eximon.conf* can be empty, but it must exist.

This is all the configuration that is needed in straightforward cases for known operating systems. However, the building process is set up so that it is easy to override options that are set by default or by operating-system-specific configuration files, for example to change the name of the C compiler, which defaults to **gcc**. See section 4.11 below for details of how to do this.

4.5 Support for iconv()

The contents of header lines in messages may be encoded according to the rules described RFC 2047. This makes it possible to transmit characters that are not in the ASCII character set, and to label them as being in a particular character set. When Exim is inspecting header lines by means of the **$h_** mechanism, it decodes them, and translates them into a specified character set (default ISO-8859-1). The translation is possible only if the operating system supports the *iconv()* function.

However, some of the operating systems that supply *iconv()* do not support very many conversions. The GNU **libiconv** library (available from **http://www.gnu.org/software/libiconv/**) can be installed on such systems to remedy this deficiency, as well as on systems that do not supply *iconv()* at all. After installing **libiconv**, you should add

```
HAVE_ICONV=yes
```

to your *Local/Makefile* and rebuild Exim.

4.6 Including TLS/SSL encryption support

Exim can be built to support encrypted SMTP connections, using the STARTTLS command as per RFC 2487. It can also support legacy clients that expect to start a TLS session immediately on connection to a non-standard port (see the **tls_on_connect_ports** runtime option and the **-tls-on-connect** command line option).

If you want to build Exim with TLS support, you must first install either the OpenSSL or GnuTLS library. There is no cryptographic code in Exim itself for implementing SSL.

If OpenSSL is installed, you should set

```
SUPPORT_TLS=yes
TLS_LIBS=-lssl -lcrypto
```

in *Local/Makefile*. You may also need to specify the locations of the OpenSSL library and include files. For example:

```
SUPPORT_TLS=yes
TLS_LIBS=-L/usr/local/openssl/lib -lssl -lcrypto
TLS_INCLUDE=-I/usr/local/openssl/include/
```

If GnuTLS is installed, you should set

```
SUPPORT_TLS=yes
USE_GNUTLS=yes
TLS_LIBS=-lgnutls -ltasn1 -lgcrypt
```

in *Local/Makefile*, and again you may need to specify the locations of the library and include files. For example:

```
SUPPORT_TLS=yes
USE_GNUTLS=yes
TLS_LIBS=-L/usr/gnu/lib -lgnutls -ltasn1 -lgcrypt
TLS_INCLUDE=-I/usr/gnu/include
```

You do not need to set TLS_INCLUDE if the relevant directory is already specified in INCLUDE. Details of how to configure Exim to make use of TLS are given in chapter 39.

4.7 Use of tcpwrappers

Exim can be linked with the *tcpwrappers* library in order to check incoming SMTP calls using the *tcpwrappers* control files. This may be a convenient alternative to Exim's own checking facilities for installations that are already making use of *tcpwrappers* for other purposes. To do this, you should set USE_TCP_WRAPPERS in *Local/Makefile*, arrange for the file *tcpd.h* to be available at compile time, and also ensure that the library *libwrap.a* is available at link time, typically by including **-lwrap** in EXTRALIBS_EXIM. For example, if *tcpwrappers* is installed in */usr/local*, you might have

```
USE_TCP_WRAPPERS=yes
CFLAGS=-O -I/usr/local/include
EXTRALIBS_EXIM=-L/usr/local/lib -lwrap
```

in *Local/Makefile*. The name to use in the *tcpwrappers* control files is "exim". For example, the line

```
exim : LOCAL  192.168.1.  .friendly.domain.example
```

in your */etc/hosts.allow* file allows connections from the local host, from the subnet 192.168.1.0/24, and from all hosts in *friendly.domain.example*. All other connections are denied. Consult the *tcpwrappers* documentation for further details.

4.8 Including support for IPv6

Exim contains code for use on systems that have IPv6 support. Setting `HAVE_IPV6=YES` in *Local/Makefile* causes the IPv6 code to be included; it may also be necessary to set IPV6_INCLUDE and IPV6_LIBS on systems where the IPv6 support is not fully integrated into the normal include and library files.

Two different types of DNS record for handling IPv6 addresses have been defined. AAAA records (analogous to A records for IPv4) are in use, and are currently seen as the mainstream. Another record type called A6 was proposed as better than AAAA because it had more flexibility. However, it was felt to be over-complex, and its status was reduced to "experimental". It is not known if anyone is actually using A6 records. Exim has support for A6 records, but this is included only if you set `SUPPORT_A6=YES` in *Local/Makefile*. The support has not been tested for some time.

4.9 The building process

Once *Local/Makefile* (and *Local/eximon.conf*, if required) have been created, run *make* at the top level. It determines the architecture and operating system types, and creates a build directory if one does not exist. For example, on a Sun system running Solaris 8, the directory *build-SunOS5-5.8-sparc* is created. Symbolic links to relevant source files are installed in the build directory.

Warning: The **-j** (parallel) flag must not be used with *make*; the building process fails if it is set.

If this is the first time *make* has been run, it calls a script that builds a make file inside the build directory, using the configuration files from the *Local* directory. The new make file is then passed to another instance of *make*. This does the real work, building a number of utility scripts, and then compiling and linking the binaries for the Exim monitor (if configured), a number of utility programs, and finally Exim itself. The command `make makefile` can be used to force a rebuild of the make file in the build directory, should this ever be necessary.

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.

4.10 Output from "make"

The output produced by the *make* process for compile lines is often very unreadable, because these lines can be very long. For this reason, the normal output is suppressed by default, and instead output similar to that which appears when compiling the 2.6 Linux kernel is generated: just a short line for each module that is being compiled or linked. However, it is still possible to get the full output, by calling *make* like this:

```
FULLECHO='' make -e
```

The value of FULLECHO defaults to "@", the flag character that suppresses command reflection in *make*. When you ask for the full output, it is given in addition to the short output.

4.11 Overriding build-time options for Exim

The main make file that is created at the beginning of the building process consists of the concatenation of a number of files which set configuration values, followed by a fixed set of *make* instructions. If a value is set more than once, the last setting overrides any previous ones. This provides a convenient way of overriding defaults. The files that are concatenated are, in order:

OS/Makefile-Default
OS/Makefile-<ostype>
Local/Makefile
Local/Makefile-<ostype>
Local/Makefile-<archtype>
Local/Makefile-<ostype>-<archtype>
OS/Makefile-Base

where *<ostype>* is the operating system type and *<archtype>* is the architecture type. *Local/Makefile* is required to exist, and the building process fails if it is absent. The other three *Local* files are optional, and are often not needed.

The values 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, thereby providing a means of forcing particular settings. Otherwise, the scripts try to get values from 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 are being used on your system.

OS/Makefile-Default contains comments about the variables that are set therein. Some (but not all) are mentioned below. If there is something that needs changing, review the contents of this file and the contents of the make file for your operating system (*OS/Makefile-<ostype>*) to see what the default values are.

If you need to change any of the values that are set in *OS/Makefile-Default* or in *OS/Makefile-<ostype>*, or to add any new definitions, you do not need to change the original files. Instead, you should make the changes by putting the new values in an appropriate *Local* file. For example, when building Exim in many releases of the Tru64-Unix (formerly Digital UNIX, formerly DEC-OSF1) operating system, it is necessary to specify that the C compiler is called *cc* rather than *gcc*. Also, the compiler must be called with the option **-std1**, to make it recognize some of the features of Standard C that Exim uses. (Most other compilers recognize Standard C by default.) To do this, you should create a file called *Local/Makefile-OSF1* containing the lines

```
CC=cc
CFLAGS=-std1
```

If you are compiling for just one operating system, it may be easier to put these lines directly into *Local/Makefile*.

Keeping all your local configuration settings separate from the distributed files makes it easy to transfer them to new versions of Exim simply by copying the contents of the *Local* directory.

Exim contains support for doing LDAP, NIS, NIS+, and other kinds of file lookup, but not all systems have these components installed, so the default is not to include the relevant code in the binary. All the different kinds of file and database lookup that Exim supports are implemented as separate code modules which are included only if the relevant compile-time options are set. In the case of LDAP, NIS, and NIS+, the settings for *Local/Makefile* are:

```
LOOKUP_LDAP=yes
LOOKUP_NIS=yes
LOOKUP_NISPLUS=yes
```

and similar settings apply to the other lookup types. They are all listed in *src/EDITME*. In many cases the relevant include files and interface libraries need to be installed before compiling Exim. However, there are some optional lookup types (such as cdb) for which the code is entirely contained within Exim, and no external include files or libraries are required. When a lookup type is not included in the binary, attempts to configure Exim to use it cause run time configuration errors.

Exim can be linked with an embedded Perl interpreter, allowing Perl subroutines to be called during string expansion. To enable this facility,

```
EXIM_PERL=perl.o
```

must be defined in *Local/Makefile*. Details of this facility are given in chapter 12.

The location of the X11 libraries is something that varies a lot between operating systems, and there may be different versions of X11 to cope with. Exim itself makes no use of X11, but if you are compiling the Exim monitor, the X11 libraries must be available. The following three variables are set in *OS/Makefile-Default*:

```
X11=/usr/X11R6
XINCLUDE=-I$(X11)/include
XLFLAGS=-L$(X11)/lib
```

These are overridden in some of the operating-system configuration files. For example, in *OS/Makefile-SunOS5* there is

```
X11=/usr/openwin
XINCLUDE=-I$(X11)/include
XLFLAGS=-L$(X11)/lib -R$(X11)/lib
```

If you need to override the default setting for your operating system, place a definition of all three of these variables into your *Local/Makefile-<ostype>* file.

If you need to add any extra libraries to the link steps, these can be put in a variable called EXTRALIBS, which appears in all the link commands, but by default is not defined. In contrast, EXTRALIBS_EXIM is used only on the command for linking the main Exim binary, and not for any associated utilities.

There is also DBMLIB, which appears in the link commands for binaries that use DBM functions (see also section 4.3). Finally, there is EXTRALIBS_EXIMON, which appears only in the link step for the Exim monitor binary, and which can be used, for example, to include additional X11 libraries.

The make file 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 non-optional file (that is, *Local/Makefile* or *Local/eximon.conf*) before rebuilding.

4.12 OS-specific header files

The *OS* directory contains a number of files with names of the form *os.h-<ostype>*. These are system-specific C header files that should not normally need to be changed. There is a list of macro

settings that are recognized in the file *OS/os.configuring*, which should be consulted if you are porting Exim to a new operating system.

4.13 Overriding build-time options for the monitor

A similar process is used for overriding things when building the Exim monitor, where the files that are involved are

OS/eximon.conf-Default
OS/eximon.conf-<ostype>
Local/eximon.conf
Local/eximon.conf-<ostype>
Local/eximon.conf-<archtype>
Local/eximon.conf-<ostype>-<archtype>

As with Exim itself, the final three files need not exist, and in this case the *OS/eximon.conf-<ostype>* file is also optional. The default values in *OS/eximon.conf-Default* can be overridden dynamically by setting environment variables of the same name, preceded by EXIMON_. For example, setting EXIMON_LOG_DEPTH in the environment overrides the value of LOG_DEPTH at run time.

4.14 Installing Exim binaries and scripts

The command `make install` runs the *exim_install* script with no arguments. The script copies binaries and utility scripts into the directory whose name is specified by the BIN_DIRECTORY setting in *Local/Makefile*. The install script copies files only if they are newer than the files they are going to replace. The Exim binary is required to be owned by root and have the *setuid* bit set, for normal configurations. Therefore, you must run `make install` as root so that it can set up the Exim binary in this way. However, in some special situations (for example, if a host is doing no local deliveries) it may be possible to run Exim without making the binary setuid root (see chapter 52 for details).

Exim's run time configuration file is named by the CONFIGURE_FILE setting in *Local/Makefile*. If this names a single file, and the file does not exist, the default configuration file *src/configure.default* is copied there by the installation script. If a run time configuration file already exists, it is left alone. If CONFIGURE_FILE is a colon-separated list, naming several alternative files, no default is installed.

One change is made to the default configuration file when it is installed: the default configuration contains a router that references a system aliases file. The path to this file is set to the value specified by SYSTEM_ALIASES_FILE in *Local/Makefile* (*/etc/aliases* by default). If the system aliases file does not exist, the installation script creates it, and outputs a comment to the user.

The created file contains no aliases, but it does contain comments about the aliases a site should normally have. Mail aliases have traditionally been kept in */etc/aliases*. However, some operating systems are now using */etc/mail/aliases*. You should check if yours is one of these, and change Exim's configuration if necessary.

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. System aliases and *.forward* files in users' home directories are supported, but no NIS or NIS+ support is configured. Domains other than the name of the local host are routed using the DNS, with delivery over SMTP.

It is possible to install Exim for special purposes (such as building a binary distribution) in a private part of the file system. You can do this by a command such as

```
make DESTDIR=/some/directory/ install
```

This has the effect of pre-pending the specified directory to all the file paths, except the name of the system aliases file that appears in the default configuration. (If a default alias file is created, its name *is* modified.) For backwards compatibility, ROOT is used if DESTDIR is not set, but this usage is deprecated.

Running *make install* does not copy the Exim 4 conversion script *convert4r4*, or the *pcretest* test program. You will probably run the first of these only once (if you are upgrading from Exim 3), and the second isn't really part of Exim. None of the documentation files in the *doc* directory are copied, except for the info files when you have set INFO_DIRECTORY, as described in section 4.15 below.

For the utility programs, old versions are renamed by adding the suffix *.O* to their names. The Exim binary itself, however, is handled differently. It is installed under a name that includes the version number and the compile number, for example *exim-4.68-1*. The script then arranges for a symbolic link called *exim* to point to the binary. If you are updating a previous version of Exim, the script takes care to ensure that the name *exim* is never absent from the directory (as seen by other processes).

If you want to see what the *make install* will do before running it for real, you can pass the **-n** option to the installation script by this command:

```
make INSTALL_ARG=-n install
```

The contents of the variable INSTALL_ARG are passed to the installation script. You do not need to be root to run this test. Alternatively, you can run the installation script directly, but this must be from within the build directory. For example, from the top-level Exim directory you could use this command:

```
(cd build-SunOS5-5.5.1-sparc; ../scripts/exim_install -n)
```

There are two other options that can be supplied to the installation script.

- **-no_chown** bypasses the call to change the owner of the installed binary to root, and the call to make it a setuid binary.
- **-no_symlink** bypasses the setting up of the symbolic link *exim* to the installed binary.

INSTALL_ARG can be used to pass these options to the script. For example:

```
make INSTALL_ARG=-no_symlink install
```

The installation script can also be given arguments specifying which files are to be copied. For example, to install just the Exim binary, and nothing else, without creating the symbolic link, you could use:

```
make INSTALL_ARG='-no_symlink exim' install
```

4.15 Installing info documentation

Not all systems use the GNU *info* system for documentation, and for this reason, the Texinfo source of Exim's documentation is not included in the main distribution. Instead it is available separately from the ftp site (see section 1.6).

If you have defined INFO_DIRECTORY in *Local/Makefile* and the Texinfo source of the documentation is found in the source tree, running `make install` automatically builds the info files and installs them.

4.16 Setting up the spool directory

When it starts up, Exim tries to create its spool directory if it does not exist. The Exim uid and gid are used for the owner and group of the spool directory. Sub-directories are automatically created in the spool directory as necessary.

4.17 Testing

Having installed Exim, you can check that the run time configuration file is syntactically valid by running the following command, which assumes that the Exim binary directory is within your PATH environment variable:

```
exim -bV
```

If there are any errors in the configuration file, Exim outputs error messages. Otherwise it outputs the version number and build date, the DBM library that is being used, and information about which drivers and other optional code modules are included in the binary. Some simple routing tests can be done by using the address testing option. For example,

```
exim -bt <local username>
```

should verify that it recognizes a local mailbox, and

```
exim -bt <remote address>
```

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 -v postmaster@your.domain.example
From: user@your.domain.example
To: postmaster@your.domain.example
Subject: Testing Exim

This is a test message.
^D
```

The **-v** option causes Exim to output some verification of what it is doing. In this case you should see copies of three log lines, one for the message's arrival, one for its delivery, and one containing "Completed".

If you encounter problems, look at Exim's log files (*mainlog* and *paniclog*) to see if there is any relevant information there. Another source of information is running Exim with debugging turned on, by specifying the **-d** option. 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 -d -M <exim-message-id>
```

You must be root or an "admin user" in order to do this. The **-d** option produces rather a lot of output, but you can cut this down to specific areas. For example, if you use **-d-all+route** only the debugging information relevant to routing is included. (See the **-d** option in chapter 5 for more details.)

One specific problem that has shown up on some sites is the inability to do local deliveries into a shared mailbox directory, because it 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). Another approach is to configure Exim not to use lock files, but just to rely on *fcntl()* locking instead. However, you should do this only if all user agents also use *fcntl()* locking. For further discussion of locking issues, see chapter 26.

One thing that cannot be tested on a system that is already running an MTA is the receipt of incoming SMTP mail on the standard SMTP port. However, the **-oX** option can be used to run an Exim daemon that listens on some other port, or *inetd* can be used to do this. The **-bh** option and the *exim_checkaccess* utility can be used to check out policy controls on incoming SMTP mail.

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 run time configuration, all other file and directory names that Exim uses can be altered, in order to keep it entirely clear of the production version.

4.18 Replacing another MTA with Exim

Building and installing Exim for the first time does not of itself put it in general use. The name by which the system's MTA is called by mail user agents is either */usr/sbin/sendmail*, or */usr/lib/sendmail* (depending on the operating system), and it is necessary to make this name point to the *exim* binary in order to get the user agents to pass messages to Exim. This is normally done by renaming any existing file and making */usr/sbin/sendmail* or */usr/lib/sendmail* a symbolic link to the

exim binary. It is a good idea to remove any setuid privilege and executable status from the old MTA. It is then necessary to stop and restart the mailer daemon, if one is running.

Some operating systems have introduced alternative ways of switching MTAs. For example, if you are running FreeBSD, you need to edit the file */etc/mail/mailer.conf* instead of setting up a symbolic link as just described. A typical example of the contents of this file for running Exim is as follows:

```
sendmail            /usr/exim/bin/exim
send-mail           /usr/exim/bin/exim
mailq               /usr/exim/bin/exim -bp
newaliases          /usr/bin/true
```

Once you have set up the symbolic link, or edited */etc/mail/mailer.conf*, your Exim installation is "live". Check it by sending a message from your favourite user agent.

You should consider what to tell your users about the change of MTA. Exim may have different capabilities to what was previously running, and there are various operational differences such as the text of messages produced by command line options and in bounce messages. If you allow your users to make use of Exim's filtering capabilities, you should make the document entitled *Exim's interface to mail filtering* available to them.

4.19 Upgrading Exim

If you are already running Exim on your host, building and installing a new version automatically makes it available to MUAs, or any other programs that call the MTA directly. However, if you are running an Exim daemon, you do need to send it a HUP signal, to make it re-execute itself, and thereby pick up the new binary. You do not need to stop processing mail in order to install a new version of Exim. The install script does not modify an existing runtime configuration file.

4.20 Stopping the Exim daemon on Solaris

The standard command for stopping the mailer daemon on Solaris is

```
/etc/init.d/sendmail stop
```

If */usr/lib/sendmail* has been turned into a symbolic link, this script fails to stop Exim because it uses the command *ps -e* and greps the output for the text "sendmail"; this is not present because the actual program name (that is, "exim") is given by the *ps* command with these options. A solution is to replace the line that finds the process id with something like

```
pid=`cat /var/spool/exim/exim-daemon.pid`
```

to obtain the daemon's pid directly from the file that Exim saves it in.

Note, however, that stopping the daemon does not "stop Exim". Messages can still be received from local processes, and if automatic delivery is configured (the normal case), deliveries will still occur.

5. The Exim command line

Exim's command line takes the standard Unix form of a sequence of options, each starting with a hyphen character, followed by a number of arguments. The options are compatible with the main options of Sendmail, and there are also some additional options, some of which are compatible with Smail 3. Certain combinations of options do not make sense, and provoke an error if used. The form of the arguments depends on which options are set.

5.1 Setting options by program name

If Exim is called under the name *mailq*, it behaves as if the option **-bp** were present before any other options. The **-bp** option requests a listing of the contents of the mail queue on the standard output. This feature is for compatibility with some systems that contain a command of that name in one of the standard libraries, symbolically linked to */usr/sbin/sendmail* or */usr/lib/sendmail*.

If Exim is called under the name *rsmtp* it behaves as if the option **-bS** were present before any other options, for compatibility with Smail. The **-bS** option is used for reading in a number of messages in batched SMTP format.

If Exim is called under the name *rmail* it behaves as if the **-i** and **-oee** options were present before any other options, for compatibility with Smail. The name *rmail* is used as an interface by some UUCP systems.

If Exim is called under the name *runq* it behaves as if the option **-q** were present before any other options, for compatibility with Smail. The **-q** option causes a single queue runner process to be started.

If Exim is called under the name *newaliases* it behaves as if the option **-bi** were present before any other options, for compatibility with Sendmail. This option is used for rebuilding Sendmail's alias file. Exim does not have the concept of a single alias file, but can be configured to run a given command if called with the **-bi** option.

5.2 Trusted and admin users

Some Exim options are available only to *trusted users* and others are available only to *admin users*. In the description below, the phrases "Exim user" and "Exim group" mean the user and group defined by EXIM_USER and EXIM_GROUP in *Local/Makefile* or set by the **exim_user** and **exim_group** options. These do not necessarily have to use the name "exim".

- The trusted users are root, the Exim user, any user listed in the **trusted_users** configuration option, and any user whose current group or any supplementary group is one of those listed in the **trusted_groups** configuration option. Note that the Exim group is not automatically trusted.

  Trusted users are always permitted to use the **-f** option or a leading "From " line to specify the envelope sender of a message that is passed to Exim through the local interface (see the **-bm** and **-f** options below). See the **untrusted_set_sender** option for a way of permitting non-trusted users to set envelope senders.

  For a trusted user, there is never any check on the contents of the *From:* header line, and a *Sender:* line is never added. Furthermore, any existing *Sender:* line in incoming local (non-TCP/IP) messages is not removed.

  Trusted users may also specify a host name, host address, interface address, protocol name, ident value, and authentication data when submitting a message locally. Thus, they are able to insert messages into Exim's queue locally that have the characteristics of messages received from a remote host. Untrusted users may in some circumstances use **-f**, but can never set the other values that are available to trusted users.

- The admin users are root, the Exim user, and any user that is a member of the Exim group or of any group listed in the **admin_groups** configuration option. The current group does not have to be one of these groups.

Admin users are permitted to list the queue, and to carry out certain operations on messages, for example, to force delivery failures. It is also necessary to be an admin user in order to see the full information provided by the Exim monitor, and full debugging output.

By default, the use of the **-M**, **-q**, **-R**, and **-S** options to cause Exim to attempt delivery of messages on its queue is restricted to admin users. However, this restriction can be relaxed by setting the **prod_requires_admin** option false (that is, specifying **no_prod_requires_admin**).

Similarly, the use of the **-bp** option to list all the messages in the queue is restricted to admin users unless **queue_list_requires_admin** is set false.

Warning: If you configure your system so that admin users are able to edit Exim's configuration file, you are giving those users an easy way of getting root. There is further discussion of this issue at the start of chapter 6.

5.3 Command line options

Exim's command line options are described in alphabetical order below. If none of the options that specifies a specific action (such as starting the daemon or a queue runner, or testing an address, or receiving a message in a specific format, or listing the queue) are present, and there is at least one argument on the command line, **-bm** (accept a local message on the standard input, with the arguments specifying the recipients) is assumed. Otherwise, Exim outputs a brief message about itself and exits.

--
This is a pseudo-option whose only purpose is to terminate the options and therefore to cause subsequent command line items to be treated as arguments rather than options, even if they begin with hyphens.

--help
This option causes Exim to output a few sentences stating what it is. The same output is generated if the Exim binary is called with no options and no arguments.

-B<*type*>
This is a Sendmail option for selecting 7 or 8 bit processing. Exim is 8-bit clean; it ignores this option.

-bd
This option runs Exim as a daemon, awaiting incoming SMTP connections. Usually the **-bd** option is combined with the **-q**<*time*> option, to specify that the daemon should also initiate periodic queue runs.

The **-bd** option can be used only by an admin user. If either of the **-d** (debugging) or **-v** (verifying) options are set, the daemon does not disconnect from the controlling terminal. When running this way, it can be stopped by pressing ctrl-C.

By default, Exim listens for incoming connections to the standard SMTP port on all the host's running interfaces. However, it is possible to listen on other ports, on multiple ports, and only on specific interfaces. Chapter 13 contains a description of the options that control this.

When a listening daemon is started without the use of **-oX** (that is, without overriding the normal configuration), it writes its process id to a file called *exim-daemon.pid* in Exim's spool directory. This location can be overridden by setting PID_FILE_PATH in *Local/Makefile*. The file is written while Exim is still running as root.

When **-oX** is used on the command line to start a listening daemon, the process id is not written to the normal pid file path. However, **-oP** can be used to specify a path on the command line if a pid file is required.

The SIGHUP signal can be used to cause the daemon to re-execute itself. This should be done whenever Exim's configuration file, or any file that is incorporated into it by means of the **.include** facility, is changed, and also whenever a new version of Exim is installed. It is not necessary to do this when other files that are referenced from the configuration (for example, alias files) are changed, because these are reread each time they are used.

-bdf
This option has the same effect as **-bd** except that it never disconnects from the controlling terminal, even when no debugging is specified.

-be
Run Exim in expansion testing mode. Exim discards its root privilege, to prevent ordinary users from using this mode to read otherwise inaccessible files. If no arguments are given, Exim runs interactively, prompting for lines of data. Otherwise, it processes each argument in turn.

If Exim was built with USE_READLINE=yes in *Local/Makefile*, it tries to load the **libreadline** library dynamically whenever the **-be** option is used without command line arguments. If successful, it uses the *readline()* function, which provides extensive line-editing facilities, for reading the test data. A line history is supported.

Long expansion expressions can be split over several lines by using backslash continuations. As in Exim's run time configuration, white space at the start of continuation lines is ignored. Each argument or data line is passed through the string expansion mechanism, and the result is output. Variable values from the configuration file (for example, *$qualify_domain*) are available, but no message-specific values (such as *$sender_domain*) are set, because no message is being processed (but see **-bem** and **-Mset**).

Note: If you use this mechanism to test lookups, and you change the data files or databases you are using, you must exit and restart Exim before trying the same lookup again. Otherwise, because each Exim process caches the results of lookups, you will just get the same result as before.

-bem <*filename*>
This option operates like **-be** except that it must be followed by the name of a file. For example:

```
exim -bem /tmp/testmessage
```

The file is read as a message (as if receiving a locally-submitted non-SMTP message) before any of the test expansions are done. Thus, message-specific variables such as *$message_size* and *$header_from:* are available. However, no *Received:* header is added to the message. If the **-t** option is set, recipients are read from the headers in the normal way, and are shown in the *$recipients* variable. Note that recipients cannot be given on the command line, because further arguments are taken as strings to expand (just like **-be**).

-bF <*filename*>
This option is the same as **-bf** except that it assumes that the filter being tested is a system filter. The additional commands that are available only in system filters are recognized.

-bf <*filename*>
This option runs Exim in user filter testing mode; the file is the filter file to be tested, and a test message must be supplied on the standard input. If there are no message-dependent tests in the filter, an empty file can be supplied.

If you want to test a system filter file, use **-bF** instead of **-bf**. You can use both **-bF** and **-bf** on the same command, in order to test a system filter and a user filter in the same run. For example:

```
exim -bF /system/filter -bf /user/filter </test/message
```

This is helpful when the system filter adds header lines or sets filter variables that are used by the user filter.

If the test filter file does not begin with one of the special lines

```
# Exim filter
# Sieve filter
```

it is taken to be a normal *.forward* file, and is tested for validity under that interpretation. See sections 22.4 to 22.6 for a description of the possible contents of non-filter redirection lists.

The result of an Exim command that uses **-bf**, provided no errors are detected, is a list of the actions that Exim would try to take if presented with the message for real. More details of filter testing are given in the separate document entitled *Exim's interfaces to mail filtering*.

When testing a filter file, the envelope sender can be set by the **-f** option, or by a "From " line at the start of the test message. Various parameters that would normally be taken from the envelope recipient address of the message can be set by means of additional command line options (see the next four options).

-bfd <*domain*>
This sets the domain of the recipient address when a filter file is being tested by means of the **-bf** option. The default is the value of *$qualify_domain*.

-bfl <*local part*>
This sets the local part of the recipient address when a filter file is being tested by means of the **-bf** option. The default is the username of the process that calls Exim. A local part should be specified with any prefix or suffix stripped, because that is how it appears to the filter when a message is actually being delivered.

-bfp <*prefix*>
This sets the prefix of the local part of the recipient address when a filter file is being tested by means of the **-bf** option. The default is an empty prefix.

-bfs <*suffix*>
This sets the suffix of the local part of the recipient address when a filter file is being tested by means of the **-bf** option. The default is an empty suffix.

-bh <*IP address*>
This option runs a fake SMTP session as if from the given IP address, using the standard input and output. The IP address may include a port number at the end, after a full stop. For example:

```
exim -bh 10.9.8.7.1234
exim -bh fe80::a00:20ff:fe86:a061.5678
```

When an IPv6 address is given, it is converted into canonical form. In the case of the second example above, the value of *$sender_host_address* after conversion to the canonical form is `fe80:0000:0000:0a00:20ff:fe86:a061.5678`.

Comments as to what is going on are written to the standard error file. These include lines beginning with "LOG" for anything that would have been logged. This facility is provided for testing configuration options for incoming messages, to make sure they implement the required policy. For example, you can test your relay controls using **-bh**.

Warning 1: You can test features of the configuration that rely on ident (RFC 1413) information by using the **-oMt** option. However, Exim cannot actually perform an ident callout when testing using **-bh** because there is no incoming SMTP connection.

Warning 2: Address verification callouts (see section 40.41) are also skipped when testing using **-bh**. If you want these callouts to occur, use **-bhc** instead.

Messages supplied during the testing session are discarded, and nothing is written to any of the real log files. There may be pauses when DNS (and other) lookups are taking place, and of course these may time out. The **-oMi** option can be used to specify a specific IP interface and port if this is important, and **-oMaa** and **-oMai** can be used to set parameters as if the SMTP session were authenticated.

The *exim_checkaccess* utility is a "packaged" version of **-bh** whose output just states whether a given recipient address from a given host is acceptable or not. See section 50.8.

Features such as authentication and encryption, where the client input is not plain text, cannot easily be tested with **-bh**. Instead, you should use a specialized SMTP test program such as **swaks (http://jetmore.org/john/code/#swaks)**.

-bhc <*IP address*>
This option operates in the same way as **-bh**, except that address verification callouts are performed if required. This includes consulting and updating the callout cache database.

-bi

Sendmail interprets the **-bi** option as a request to rebuild its alias file. Exim does not have the concept of a single alias file, and so it cannot mimic this behaviour. However, calls to */usr/lib/sendmail* with the **-bi** option tend to appear in various scripts such as NIS make files, so the option must be recognized.

If **-bi** is encountered, the command specified by the **bi_command** configuration option is run, under the uid and gid of the caller of Exim. If the **-oA** option is used, its value is passed to the command as an argument. The command set by **bi_command** may not contain arguments. The command can use the *exim_dbmbuild* utility, or some other means, to rebuild alias files if this is required. If the **bi_command** option is not set, calling Exim with **-bi** is a no-op.

-bm

This option runs an Exim receiving process that accepts an incoming, locally-generated message on the current input. The recipients are given as the command arguments (except when **-t** is also present – see below). Each argument can be a comma-separated list of RFC 2822 addresses. This is the default option for selecting the overall action of an Exim call; it is assumed if no other conflicting option is present.

If any addresses in the message are unqualified (have no domain), they are qualified by the values of the **qualify_domain** or **qualify_recipient** options, as appropriate. The **-bnq** option (see below) provides a way of suppressing this for special cases.

Policy checks on the contents of local messages can be enforced by means of the non-SMTP ACL. See chapter 40 for details.

The return code is zero if the message is successfully accepted. Otherwise, the action is controlled by the **-oe***x* option setting – see below.

The format of the message must be as defined in RFC 2822, except that, for compatibility with Sendmail and Smail, a line in one of the forms

```
From sender Fri Jan  5 12:55 GMT 1997
From sender Fri, 5 Jan 97 12:55:01
```

(with the weekday optional, and possibly with additional text after the date) is permitted to appear at the start of the message. There appears to be no authoritative specification of the format of this line. Exim recognizes it by matching against the regular expression defined by the **uucp_from_pattern** option, which can be changed if necessary.

The specified sender is treated as if it were given as the argument to the **-f** option, but if a **-f** option is also present, its argument is used in preference to the address taken from the message. The caller of Exim must be a trusted user for the sender of a message to be set in this way.

-bnq

By default, Exim automatically qualifies unqualified addresses (those without domains) that appear in messages that are submitted locally (that is, not over TCP/IP). This qualification applies both to addresses in envelopes, and addresses in header lines. Sender addresses are qualified using **qualify_domain**, and recipient addresses using **qualify_recipient** (which defaults to the value of **qualify_domain**).

Sometimes, qualification is not wanted. For example, if **-bS** (batch SMTP) is being used to re-submit messages that originally came from remote hosts after content scanning, you probably do not want to qualify unqualified addresses in header lines. (Such lines will be present only if you have not enabled a header syntax check in the appropriate ACL.)

The **-bnq** option suppresses all qualification of unqualified addresses in messages that originate on the local host. When this is used, unqualified addresses in the envelope provoke errors (causing message rejection) and unqualified addresses in header lines are left alone.

-bP

If this option is given with no arguments, it causes the values of all Exim's main configuration options to be written to the standard output. The values of one or more specific options can be requested by giving their names as arguments, for example:

```
exim -bP qualify_domain hold_domains
```

However, any option setting that is preceded by the word "hide" in the configuration file is not shown in full, except to an admin user. For other users, the output is as in this example:

```
mysql_servers = <value not displayable>
```

If **configure_file** is given as an argument, the name of the run time configuration file is output. If a list of configuration files was supplied, the value that is output here is the name of the file that was actually used.

If **log_file_path** or **pid_file_path** are given, the names of the directories where log files and daemon pid files are written are output, respectively. If these values are unset, log files are written in a sub-directory of the spool directory called **log**, and the pid file is written directly into the spool directory.

If **-bP** is followed by a name preceded by +, for example,

```
exim -bP +local_domains
```

it searches for a matching named list of any type (domain, host, address, or local part) and outputs what it finds.

If one of the words **router**, **transport**, or **authenticator** is given, followed by the name of an appropriate driver instance, the option settings for that driver are output. For example:

```
exim -bP transport local_delivery
```

The generic driver options are output first, followed by the driver's private options. A list of the names of drivers of a particular type can be obtained by using one of the words **router_list**, **transport_list**, or **authenticator_list**, and a complete list of all drivers with their option settings can be obtained by using **routers**, **transports**, or **authenticators**.

-bp
This option requests a listing of the contents of the mail queue on the standard output. If the **-bp** option is followed by a list of message ids, just those messages are listed. By default, this option can be used only by an admin user. However, the **queue_list_requires_admin** option can be set false to allow any user to see the queue.

Each message on the queue is displayed as in the following example:

```
25m  2.9K 0t5C6f-0000c8-00 <alice@wonderland.fict.example>
           red.king@looking-glass.fict.example
           <other addresses>
```

The first line contains the length of time the message has been on the queue (in this case 25 minutes), the size of the message (2.9K), the unique local identifier for the message, and the message sender, as contained in the envelope. For bounce messages, the sender address is empty, and appears as "<>". If the message was submitted locally by an untrusted user who overrode the default sender address, the user's login name is shown in parentheses before the sender address.

If the message is frozen (attempts to deliver it are suspended) then the text "*** frozen ***" is displayed at the end of this line.

The recipients of the message (taken from the envelope, not the headers) are displayed on subsequent lines. Those addresses to which the message has already been delivered are marked with the letter D. If an original address gets expanded into several addresses via an alias or forward file, the original is displayed with a D only when deliveries for all of its child addresses are complete.

-bpa
This option operates like **-bp**, but in addition it shows delivered addresses that were generated from the original top level address(es) in each message by alias or forwarding operations. These addresses are flagged with "+D" instead of just "D".

-bpc
This option counts the number of messages on the queue, and writes the total to the standard output. It is restricted to admin users, unless **queue_list_requires_admin** is set false.

-bpr
This option operates like **-bp**, but the output is not sorted into chronological order of message arrival. This can speed it up when there are lots of messages on the queue, and is particularly useful if the output is going to be post-processed in a way that doesn't need the sorting.

-bpra
This option is a combination of **-bpr** and **-bpa**.

-bpru
This option is a combination of **-bpr** and **-bpu**.

-bpu
This option operates like **-bp** but shows only undelivered top-level addresses for each message displayed. Addresses generated by aliasing or forwarding are not shown, unless the message was deferred after processing by a router with the **one_time** option set.

-brt
This option is for testing retry rules, and it must be followed by up to three arguments. It causes Exim to look for a retry rule that matches the values and to write it to the standard output. For example:

```
exim -brt bach.comp.mus.example
Retry rule: *.comp.mus.example  F,2h,15m; F,4d,30m;
```

See chapter 32 for a description of Exim's retry rules. The first argument, which is required, can be a complete address in the form *local_part@domain*, or it can be just a domain name. If the second argument contains a dot, it is interpreted as an optional second domain name; if no retry rule is found for the first argument, the second is tried. This ties in with Exim's behaviour 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. Finally, an argument that is the name of a specific delivery error, as used in setting up retry rules, can be given. For example:

```
exim -brt haydn.comp.mus.example quota_3d
Retry rule: *@haydn.comp.mus.example quota_3d  F,1h,15m
```

-brw
This option is for testing address rewriting rules, and it must be followed by a single argument, consisting of either a local part without a domain, or a complete address with a fully qualified domain. Exim outputs how this address would be rewritten for each possible place it might appear. See chapter 31 for further details.

-bS
This option is used for batched SMTP input, which is an alternative interface for non-interactive local message submission. A number of messages can be submitted in a single run. However, despite its name, this is not really SMTP input. Exim reads each message's envelope from SMTP commands on the standard input, but generates no responses. If the caller is trusted, or **untrusted_set_sender** is set, the senders in the SMTP MAIL commands are believed; otherwise the sender is always the caller of Exim.

The message itself is read from the standard input, in SMTP format (leading dots doubled), terminated by a line containing just a single dot. An error is provoked if the terminating dot is missing. A further message may then follow.

As for other local message submissions, the contents of incoming batch SMTP messages can be checked using the non-SMTP ACL (see chapter 40). Unqualified addresses are automatically qualified using **qualify_domain** and **qualify_recipient**, as appropriate, unless the **-bnq** option is used.

Some other SMTP commands are recognized in the input. HELO and EHLO act as RSET; VRFY, EXPN, ETRN, and HELP act as NOOP; QUIT quits, ignoring the rest of the standard input.

If any error is encountered, reports are written to the standard output and error streams, and Exim gives up immediately. The return code is 0 if no error was detected; it is 1 if one or more messages were accepted before the error was detected; otherwise it is 2.

More details of input using batched SMTP are given in section 45.11.

-bs

This option causes Exim to accept one or more messages by reading SMTP commands on the standard input, and producing SMTP replies on the standard output. SMTP policy controls, as defined in ACLs (see chapter 40) are applied. Some user agents use this interface as a way of passing locally-generated messages to the MTA.

In this usage, if the caller of Exim is trusted, or **untrusted_set_sender** is set, the senders of messages are taken from the SMTP MAIL commands. Otherwise the content of these commands is ignored and the sender is set up as the calling user. Unqualified addresses are automatically qualified using **qualify_domain** and **qualify_recipient**, as appropriate, unless the **-bnq** option is used.

The **-bs** option is also used to run Exim from *inetd*, as an alternative to using a listening daemon. Exim can distinguish the two cases by checking whether the standard input is a TCP/IP socket. When Exim is called from *inetd*, the source of the mail is assumed to be remote, and the comments above concerning senders and qualification do not apply. In this situation, Exim behaves in exactly the same way as it does when receiving a message via the listening daemon.

-bt

This option runs Exim in address testing mode, in which each argument is taken as a recipient address to be tested for deliverability. The results are written to the standard output. If a test fails, and the caller is not an admin user, no details of the failure are output, because these might contain sensitive information such as usernames and passwords for database lookups.

If no arguments are given, Exim runs in an interactive manner, prompting with a right angle bracket for addresses to be tested.

Unlike the **-be** test option, you cannot arrange for Exim to use the *readline()* function, because it is running as *root* and there are security issues.

Each address is handled as if it were the recipient address of a message (compare the **-bv** option). It is passed to the routers and the result is written to the standard output. However, any router that has **no_address_test** set is bypassed. This can make **-bt** easier to use for genuine routing tests if your first router passes everything to a scanner program.

The return code is 2 if any address failed outright; it is 1 if no address failed outright but at least one could not be resolved for some reason. Return code 0 is given only when all addresses succeed.

Note: When actually delivering a message, Exim removes duplicate recipient addresses after routing is complete, so that only one delivery takes place. This does not happen when testing with **-bt**; the full results of routing are always shown.

Warning: **-bt** can only do relatively simple testing. If any of the routers in the configuration makes any tests on the sender address of a message, you can use the **-f** option to set an appropriate sender when running **-bt** tests. Without it, the sender is assumed to be the calling user at the default qualifying domain. However, if you have set up (for example) routers whose behaviour depends on the contents of an incoming message, you cannot test those conditions using **-bt**. The **-N** option provides a possible way of doing such tests.

-bV

This option causes Exim to write the current version number, compilation number, and compilation date of the *exim* binary to the standard output. It also lists the DBM library this is being used, the optional modules (such as specific lookup types), the drivers that are included in the binary, and the name of the run time configuration file that is in use.

As part of its operation, **-bV** causes Exim to read and syntax check its configuration file. However, this is a static check only. It cannot check values that are to be expanded. For example, although a misspelt ACL verb is detected, an error in the verb's arguments is not. You cannot rely on **-bV** alone to discover (for example) all the typos in the configuration; some realistic testing is needed. The **-bh** and **-N** options provide more dynamic testing facilities.

-bv

This option runs Exim in address verification mode, in which each argument is taken as a recipient address to be verified by the routers. (This does not involve any verification callouts). During normal operation, verification happens mostly as a consequence processing a **verify** condition in an ACL (see chapter 40). If you want to test an entire ACL, possibly including callouts, see the **-bh** and **-bhc** options.

If verification fails, and the caller is not an admin user, no details of the failure are output, because these might contain sensitive information such as usernames and passwords for database lookups.

If no arguments are given, Exim runs in an interactive manner, prompting with a right angle bracket for addresses to be verified.

Unlike the **-be** test option, you cannot arrange for Exim to use the *readline()* function, because it is running as *exim* and there are security issues.

Verification differs from address testing (the **-bt** option) in that routers that have **no_verify** set are skipped, and if the address is accepted by a router that has **fail_verify** set, verification fails. The address is verified as a recipient if **-bv** is used; to test verification for a sender address, **-bvs** should be used.

If the **-v** option is not set, the output consists of a single line for each address, stating whether it was verified or not, and giving a reason in the latter case. Without **-v**, generating more than one address by redirection causes verification to end successfully, without considering the generated addresses. However, if just one address is generated, processing continues, and the generated address must verify successfully for the overall verification to succeed.

When **-v** is set, more details are given of how the address has been handled, and in the case of address redirection, all the generated addresses are also considered. Verification may succeed for some and fail for others.

The return code is 2 if any address failed outright; it is 1 if no address failed outright but at least one could not be resolved for some reason. Return code 0 is given only when all addresses succeed.

If any of the routers in the configuration makes any tests on the sender address of a message, you should use the **-f** option to set an appropriate sender when running **-bv** tests. Without it, the sender is assumed to be the calling user at the default qualifying domain.

-bvs

This option acts like **-bv**, but verifies the address as a sender rather than a recipient address. This affects any rewriting and qualification that might happen.

-C <*filelist*>

This option causes Exim to find the run time configuration file from the given list instead of from the list specified by the CONFIGURE_FILE compile-time setting. Usually, the list will consist of just a single file name, but it can be a colon-separated list of names. In this case, the first file that exists is used. Failure to open an existing file stops Exim from proceeding any further along the list, and an error is generated.

When this option is used by a caller other than root or the Exim user, and the list is different from the compiled-in list, Exim gives up its root privilege immediately, and runs with the real and effective uid and gid set to those of the caller. However, if ALT_CONFIG_ROOT_ONLY is defined in *Local/Makefile*, root privilege is retained for **-C** only if the caller of Exim is root.

That is, the Exim user is no longer privileged in this regard. This build-time option is not set by default in the Exim source distribution tarbundle. However, if you are using a "packaged" version of Exim (source or binary), the packagers might have enabled it.

Setting ALT_CONFIG_ROOT_ONLY locks out the possibility of testing a configuration using **-C** right through message reception and delivery, even if the caller is root. The reception works, but by that time, Exim is running as the Exim user, so when it re-executes to regain privilege for the delivery, the use of **-C** causes privilege to be lost. However, root can test reception and delivery

using two separate commands (one to put a message on the queue, using **-odq**, and another to do the delivery, using **-M**).

If ALT_CONFIG_PREFIX is defined *in Local/Makefile*, it specifies a prefix string with which any file named in a **-C** command line option must start. In addition, the file name must not contain the sequence `/../`. However, if the value of the **-C** option is identical to the value of CONFIGURE_FILE in *Local/Makefile*, Exim ignores **-C** and proceeds as usual. There is no default setting for ALT_CONFIG_PREFIX; when it is unset, any file name can be used with **-C**.

ALT_CONFIG_PREFIX can be used to confine alternative configuration files to a directory to which only root has access. This prevents someone who has broken into the Exim account from running a privileged Exim with an arbitrary configuration file.

The **-C** facility is useful for ensuring that configuration files are syntactically correct, but cannot be used for test deliveries, unless the caller is privileged, or unless it is an exotic configuration that does not require privilege. No check is made on the owner or group of the files specified by this option.

-D<*macro*>=<*value*>
This option can be used to override macro definitions in the configuration file (see section 6.4). However, like **-C**, if it is used by an unprivileged caller, it causes Exim to give up its root privilege. If DISABLE_D_OPTION is defined in *Local/Makefile*, the use of **-D** is completely disabled, and its use causes an immediate error exit.

The entire option (including equals sign if present) must all be within one command line item. **-D** can be used to set the value of a macro to the empty string, in which case the equals sign is optional. These two commands are synonymous:

```
exim -DABC  ...
exim -DABC= ...
```

To include spaces in a macro definition item, quotes must be used. If you use quotes, spaces are permitted around the macro name and the equals sign. For example:

```
exim '-D ABC = something' ...
```

-D may be repeated up to 10 times on a command line.

-d<*debug options*>
This option causes debugging information to be written to the standard error stream. It is restricted to admin users because debugging output may show database queries that contain password information. Also, the details of users' filter files should be protected. If a non-admin user uses **-d**, Exim writes an error message to the standard error stream and exits with a non-zero return code.

When **-d** is used, **-v** is assumed. If **-d** is given on its own, a lot of standard debugging data is output. This can be reduced, or increased to include some more rarely needed information, by directly following **-d** with a string made up of names preceded by plus or minus characters. These add or remove sets of debugging data, respectively. For example, **-d+filter** adds filter debugging, whereas **-d-all+filter** selects only filter debugging. Note that no spaces are allowed in the debug setting. The available debugging categories are:

`acl`	ACL interpretation
`auth`	authenticators
`deliver`	general delivery logic
`dns`	DNS lookups (see also resolver)
`dnsbl`	DNS black list (aka RBL) code
`exec`	arguments for *execv()* calls
`expand`	detailed debugging for string expansions
`filter`	filter handling
`hints_lookup`	hints data lookups
`host_lookup`	all types of name-to-IP address handling
`ident`	ident lookup
`interface`	lists of local interfaces
`lists`	matching things in lists

`load`	system load checks
`local_scan`	can be used by *local_scan()* (see chapter 42)
`lookup`	general lookup code and all lookups
`memory`	memory handling
`pid`	add pid to debug output lines
`process_info`	setting info for the process log
`queue_run`	queue runs
`receive`	general message reception logic
`resolver`	turn on the DNS resolver's debugging output
`retry`	retry handling
`rewrite`	address rewriting
`route`	address routing
`timestamp`	add timestamp to debug output lines
`tls`	TLS logic
`transport`	transports
`uid`	changes of uid/gid and looking up uid/gid
`verify`	address verification logic
`all`	almost all of the above (see below), and also **-v**

The `all` option excludes `memory` when used as `+all`, but includes it for `-all`. The reason for this is that `+all` is something that people tend to use when generating debug output for Exim maintainers. If `+memory` is included, an awful lot of output that is very rarely of interest is generated, so it now has to be explicitly requested. However, `-all` does turn everything off.

The `resolver` option produces output only if the DNS resolver was compiled with DEBUG enabled. This is not the case in some operating systems. Also, unfortunately, debugging output from the DNS resolver is written to stdout rather than stderr.

The default (**-d** with no argument) omits `expand`, `filter`, `interface`, `load`, `memory`, `pid`, `resolver`, and `timestamp`. However, the `pid` selector is forced when debugging is turned on for a daemon, which then passes it on to any re-executed Exims. Exim also automatically adds the pid to debug lines when several remote deliveries are run in parallel.

The `timestamp` selector causes the current time to be inserted at the start of all debug output lines. This can be useful when trying to track down delays in processing.

If the **debug_print** option is set in any driver, it produces output whenever any debugging is selected, or if **-v** is used.

-dd<*debug options*>
This option behaves exactly like **-d** except when used on a command that starts a daemon process. In that case, debugging is turned off for the subprocesses that the daemon creates. Thus, it is useful for monitoring the behaviour of the daemon without creating as much output as full debugging does.

-dropcr
This is an obsolete option that is now a no-op. It used to affect the way Exim handled CR and LF characters in incoming messages. What happens now is described in section 44.2.

-E
This option specifies that an incoming message is a locally-generated delivery failure report. It is used internally by Exim when handling delivery failures and is not intended for external use. Its only effect is to stop Exim generating certain messages to the postmaster, as otherwise message cascades could occur in some situations. As part of the same option, a message id may follow the characters **-E**. If it does, the log entry for the receipt of the new message contains the id, following "R=", as a cross-reference.

-e*x*
There are a number of Sendmail options starting with **-oe** which seem to be called by various programs without the leading **o** in the option. For example, the **vacation** program uses **-eq**. Exim treats all options of the form **-e***x* as synonymous with the corresponding **-oe***x* options.

-F <*string*>
This option sets the sender's full name for use when a locally-generated message is being accepted. In the absence of this option, the user's *gecos* entry from the password data is used. As users are generally permitted to alter their *gecos* entries, no security considerations are involved. White space between **-F** and the <*string*> is optional.

-f <*address*>
This option sets the address of the envelope sender of a locally-generated message (also known as the return path). The option can normally be used only by a trusted user, but **untrusted_set_sender** can be set to allow untrusted users to use it.

Processes running as root or the Exim user are always trusted. Other trusted users are defined by the **trusted_users** or **trusted_groups** options. In the absence of **-f**, or if the caller is not trusted, the sender of a local message is set to the caller's login name at the default qualify domain.

There is one exception to the restriction on the use of **-f**: an empty sender can be specified by any user, trusted or not, to create a message that can never provoke a bounce. An empty sender can be specified either as an empty string, or as a pair of angle brackets with nothing between them, as in these examples of shell commands:

```
exim -f '<>' user@domain
exim -f "" user@domain
```

In addition, the use of **-f** is not restricted when testing a filter file with **-bf** or when testing or verifying addresses using the **-bt** or **-bv** options.

Allowing untrusted users to change the sender address does not of itself make it possible to send anonymous mail. Exim still checks that the *From:* header refers to the local user, and if it does not, it adds a *Sender:* header, though this can be overridden by setting **no_local_from_check**.

White space between **-f** and the <*address*> is optional (that is, they can be given as two arguments or one combined argument). The sender of a locally-generated message can also be set (when permitted) by an initial "From " line in the message – see the description of **-bm** above – but if **-f** is also present, it overrides "From ".

-G
This is a Sendmail option which is ignored by Exim.

-h <*number*>
This option is accepted for compatibility with Sendmail, but has no effect. (In Sendmail it overrides the "hop count" obtained by counting *Received:* headers.)

-i
This option, which has the same effect as **-oi**, specifies that a dot on a line by itself should not terminate an incoming, non-SMTP message. I can find no documentation for this option in Solaris 2.4 Sendmail, but the *mailx* command in Solaris 2.4 uses it. See also **-ti**.

-M <*message id*> <*message id*> ...
This option requests Exim to run a delivery attempt on each message in turn. If any of the messages are frozen, they are automatically thawed before the delivery attempt. The settings of **queue_domains**, **queue_smtp_domains**, and **hold_domains** are ignored.

Retry hints for any of the addresses are overridden – Exim tries to deliver even if the normal retry time has not yet been reached. This option requires the caller to be an admin user. However, there is an option called **prod_requires_admin** which can be set false to relax this restriction (and also the same requirement for the **-q**, **-R**, and **-S** options).

The deliveries happen synchronously, that is, the original Exim process does not terminate until all the delivery attempts have finished. No output is produced unless there is a serious error. If you want to see what is happening, use the **-v** option as well, or inspect Exim's main log.

-Mar <*message id*> <*address*> <*address*> ...
This option requests Exim to add the addresses to the list of recipients of the message ("ar" for "add recipients"). The first argument must be a message id, and the remaining ones must be email

addresses. However, if the message is active (in the middle of a delivery attempt), it is not altered. This option can be used only by an admin user.

-MC <*transport*> <*hostname*> <*sequence number*> <*message id*>
This option is not intended for use by external callers. It is used internally by Exim to invoke another instance of itself to deliver a waiting message using an existing SMTP connection, which is passed as the standard input. Details are given in chapter 45. This must be the final option, and the caller must be root or the Exim user in order to use it.

-MCA
This option is not intended for use by external callers. It is used internally by Exim in conjunction with the **-MC** option. It signifies that the connection to the remote host has been authenticated.

-MCP
This option is not intended for use by external callers. It is used internally by Exim in conjunction with the **-MC** option. It signifies that the server to which Exim is connected supports pipelining.

-MCQ <*process id*> <*pipe fd*>
This option is not intended for use by external callers. It is used internally by Exim in conjunction with the **-MC** option when the original delivery was started by a queue runner. It passes on the process id of the queue runner, together with the file descriptor number of an open pipe. Closure of the pipe signals the final completion of the sequence of processes that are passing messages through the same SMTP connection.

-MCS
This option is not intended for use by external callers. It is used internally by Exim in conjunction with the **-MC** option, and passes on the fact that the SMTP SIZE option should be used on messages delivered down the existing connection.

-MCT
This option is not intended for use by external callers. It is used internally by Exim in conjunction with the **-MC** option, and passes on the fact that the host to which Exim is connected supports TLS encryption.

-Mc <*message id*> <*message id*> ...
This option requests Exim to run a delivery attempt on each message in turn, but unlike the **-M** option, it does check for retry hints, and respects any that are found. This option is not very useful to external callers. It is provided mainly for internal use by Exim when it needs to re-invoke itself in order to regain root privilege for a delivery (see chapter 52). However, **-Mc** can be useful when testing, in order to run a delivery that respects retry times and other options such as **hold_domains** that are overridden when **-M** is used. Such a delivery does not count as a queue run. If you want to run a specific delivery as if in a queue run, you should use **-q** with a message id argument. A distinction between queue run deliveries and other deliveries is made in one or two places.

-Mes <*message id*> <*address*>
This option requests Exim to change the sender address in the message to the given address, which must be a fully qualified address or "<>" ("es" for "edit sender"). There must be exactly two arguments. The first argument must be a message id, and the second one an email address. However, if the message is active (in the middle of a delivery attempt), its status is not altered. This option can be used only by an admin user.

-Mf <*message id*> <*message id*> ...
This option requests Exim to mark each listed message as "frozen". This prevents any delivery attempts taking place until the message is "thawed", either manually or as a result of the **auto_thaw** configuration option. However, if any of the messages are active (in the middle of a delivery attempt), their status is not altered. This option can be used only by an admin user.

-Mg <*message id*> <*message id*> ...
This option requests Exim to give up trying to deliver the listed messages, including any that are frozen. However, if any of the messages are active, their status is not altered. For non-bounce messages, a delivery error message is sent to the sender, containing the text "cancelled by administrator". Bounce messages are just discarded. This option can be used only by an admin user.

-Mmad <*message id*> <*message id*> ...
This option requests Exim to mark all the recipient addresses in the messages as already delivered ("mad" for "mark all delivered"). However, if any message is active (in the middle of a delivery attempt), its status is not altered. This option can be used only by an admin user.

-Mmd <*message id*> <*address*> <*address*> ...
This option requests Exim to mark the given addresses as already delivered ("md" for "mark delivered"). The first argument must be a message id, and the remaining ones must be email addresses. These are matched to recipient addresses in the message in a case-sensitive manner. If the message is active (in the middle of a delivery attempt), its status is not altered. This option can be used only by an admin user.

-Mrm <*message id*> <*message id*> ...
This option requests Exim to remove the given messages from the queue. No bounce messages are sent; each message is simply forgotten. However, if any of the messages are active, their status is not altered. This option can be used only by an admin user or by the user who originally caused the message to be placed on the queue.

-Mset <*message id*>
This option is useful only in conjunction with **-be** (that is, when testing string expansions). Exim loads the given message from its spool before doing the test expansions, thus setting message-specific variables such as *$message_size* and the header variables. The *$recipients* variable is made available. This feature is provided to make it easier to test expansions that make use of these variables. However, this option can be used only by an admin user. See also **-bem**.

-Mt <*message id*> <*message id*> ...
This option requests Exim to "thaw" any of the listed messages that are "frozen", so that delivery attempts can resume. However, if any of the messages are active, their status is not altered. This option can be used only by an admin user.

-Mvb <*message id*>
This option causes the contents of the message body (-D) spool file to be written to the standard output. This option can be used only by an admin user.

-Mvc <*message id*>
This option causes a copy of the complete message (header lines plus body) to be written to the standard output in RFC 2822 format. This option can be used only by an admin user.

-Mvh <*message id*>
This option causes the contents of the message headers (-H) spool file to be written to the standard output. This option can be used only by an admin user.

-Mvl <*message id*>
This option causes the contents of the message log spool file to be written to the standard output. This option can be used only by an admin user.

-m
This is apparently a synonym for **-om** that is accepted by Sendmail, so Exim treats it that way too.

-N
This is a debugging option that inhibits delivery of a message at the transport level. It implies **-v**. Exim goes through many of the motions of delivery – it just doesn't actually transport the message, but instead 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 "=>".

Because **-N** discards any message to which it applies, only root or the Exim user are allowed to use it with **-bd**, **-q**, **-R** or **-M**. In other words, an ordinary user can use it only when supplying an incoming message to which it will apply. Although transportation never fails when **-N** is set, an address may be deferred because of a configuration problem on a transport, or a routing problem. Once **-N** has been used for a delivery attempt, it sticks to the message, and applies to any subsequent delivery attempts that may happen for that message.

-n
This option is interpreted by Sendmail to mean "no aliasing". It is ignored by Exim.

-O <*data*>
This option is interpreted by Sendmail to mean `set option`. It is ignored by Exim.

-oA <*file name*>
This option is used by Sendmail in conjunction with **-bi** to specify an alternative alias file name. Exim handles **-bi** differently; see the description above.

-oB <*n*>
This is a debugging option which limits the maximum number of messages that can be delivered down one SMTP connection, overriding the value set in any *smtp* transport. If <*n*> is omitted, the limit is set to 1.

-odb
This 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 does not wait for the delivery processes to finish.

When all the messages have been received, the reception process exits, leaving the delivery processes to finish in their own time. The standard output and error streams are closed at the start of each delivery process. This is the default action if none of the **-od** options are present.

If one of the queueing options in the configuration file (**queue_only** or **queue_only_file**, for example) is in effect, **-odb** overrides it if **queue_only_override** is set true, which is the default setting. If **queue_only_override** is set false, **-odb** has no effect.

-odf
This option requests "foreground" (synchronous) delivery when Exim has accepted a locally-generated message. (For the daemon it is exactly the same as **-odb**.) A delivery process is automatically started to deliver the message, and Exim waits for it to complete before proceeding.

The original Exim reception process does not finish until the delivery process for the final message has ended. The standard error stream is left open during deliveries.

However, like **-odb**, this option has no effect if **queue_only_override** is false and one of the queueing options in the configuration file is in effect.

If there is a temporary delivery error during foreground delivery, the message is left on the queue for later delivery, and the original reception process exits. See chapter 48 for a way of setting up a restricted configuration that never queues messages.

-odi
This option is synonymous with **-odf**. It is provided for compatibility with Sendmail.

-odq
This 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 process for each message received. Messages are placed on the queue, and remain there until a subsequent queue runner process encounters them. There are several configuration options (such as **queue_only**) that can be used to queue incoming messages under certain conditions. This option overrides all of them and also **-odqs**. It always forces queueing.

-odqs
This option is a hybrid between **-odb/-odi** and **-odq**. However, like **-odb** and **-odi**, this option has no effect if **queue_only_override** is false and one of the queueing options in the configuration file is in effect.

When **-odqs** does operate, a delivery process is started for each incoming message, in the background by default, but in the foreground if **-odi** is also present. The recipient addresses are routed, and local deliveries are done in the normal way. However, if any SMTP deliveries are required, they are not done at this time, so the message remains on the queue until a subsequent queue runner process encounters it. Because routing was done, Exim knows which messages are waiting for which hosts, and so a number of messages for the same host can be sent in a single SMTP

connection. The **queue_smtp_domains** configuration option has the same effect for specific domains. See also the **-qq** option.

-oee
If an error is detected while a non-SMTP message is being received (for example, a malformed address), the error is reported to the sender in a mail message.

Provided this error message is successfully sent, the Exim receiving process exits with a return code of zero. If not, the return code is 2 if the problem is that the original message has no recipients, or 1 any other error. This is the default **-oe***x* option if Exim is called as *rmail*.

-oem
This is the same as **-oee**, except that Exim always exits with a non-zero return code, whether or not the error message was successfully sent. This is the default **-oe***x* option, unless Exim is called as *rmail*.

-oep
If an error is detected while a non-SMTP message is being received, the error is reported by writing a message to the standard error file (stderr). The return code is 1 for all errors.

-oeq
This option is supported for compatibility with Sendmail, but has the same effect as **-oep**.

-oew
This option is supported for compatibility with Sendmail, but has the same effect as **-oem**.

-oi
This option, which has the same effect as **-i**, specifies that a dot on a line by itself should not terminate an incoming, non-SMTP message. Otherwise, a single dot does terminate, though Exim does no special processing for other lines that start with a dot. This option is set by default if Exim is called as *rmail*. See also **-ti**.

-oitrue
This option is treated as synonymous with **-oi**.

-oMa <*host address*>
A number of options starting with **-oM** can be used to set values associated with remote hosts on locally-submitted messages (that is, messages not received over TCP/IP). These options can be used by any caller in conjunction with the **-bh**, **-be**, **-bf**, **-bF**, **-bt**, or **-bv** testing options. In other circumstances, they are ignored unless the caller is trusted.

The **-oMa** option sets the sender host address. This may include a port number at the end, after a full stop (period). For example:

```
exim -bs -oMa 10.9.8.7.1234
```

An alternative syntax is to enclose the IP address in square brackets, followed by a colon and the port number:

```
exim -bs -oMa [10.9.8.7]:1234
```

The IP address is placed in the *$sender_host_address* variable, and the port, if present, in *$sender_host_port*. If both **-oMa** and **-bh** are present on the command line, the sender host IP address is taken from whichever one is last.

-oMaa <*name*>
See **-oMa** above for general remarks about the **-oM** options. The **-oMaa** option sets the value of *$sender_host_authenticated* (the authenticator name). See chapter 33 for a discussion of SMTP authentication. This option can be used with **-bh** and **-bs** to set up an authenticated SMTP session without actually using the SMTP AUTH command.

-oMai <*string*>
See **-oMa** above for general remarks about the **-oM** options. The **-oMai** option sets the value of *$authenticated_id* (the id that was authenticated). This overrides the default value (the caller's login id, except with **-bh**, where there is no default) for messages from local sources. See chapter 33 for a discussion of authenticated ids.

-oMas <*address*>
See **-oMa** above for general remarks about the **-oM** options. The **-oMas** option sets the authenticated sender value in *$authenticated_sender*. It overrides the sender address that is created from the caller's login id for messages from local sources, except when **-bh** is used, when there is no default. For both **-bh** and **-bs**, an authenticated sender that is specified on a MAIL command overrides this value. See chapter 33 for a discussion of authenticated senders.

-oMi <*interface address*>
See **-oMa** above for general remarks about the **-oM** options. The **-oMi** option sets the IP interface address value. A port number may be included, using the same syntax as for **-oMa**. The interface address is placed in *$received_ip_address* and the port number, if present, in *$received_port*.

-oMr <*protocol name*>
See **-oMa** above for general remarks about the **-oM** options. The **-oMr** option sets the received protocol value that is stored in *$received_protocol*. However, it does not apply (and is ignored) when **-bh** or **-bs** is used. For **-bh**, the protocol is forced to one of the standard SMTP protocol names (see the description of *$received_protocol* in section 11.9). For **-bs**, the protocol is always "local-" followed by one of those same names. For **-bS** (batched SMTP) however, the protocol can be set by **-oMr**.

-oMs <*host name*>
See **-oMa** above for general remarks about the **-oM** options. The **-oMs** option sets the sender host name in *$sender_host_name*. When this option is present, Exim does not attempt to look up a host name from an IP address; it uses the name it is given.

-oMt <*ident string*>
See **-oMa** above for general remarks about the **-oM** options. The **-oMt** option sets the sender ident value in *$sender_ident*. The default setting for local callers is the login id of the calling process, except when **-bh** is used, when there is no default.

-om
In Sendmail, this option means "me too", indicating that the sender of a message should receive a copy of the message if the sender appears in an alias expansion. Exim always does this, so the option does nothing.

-oo
This option is ignored. In Sendmail it specifies "old style headers", whatever that means.

-oP <*path*>
This option is useful only in conjunction with **-bd** or **-q** with a time value. The option specifies the file to which the process id of the daemon is written. When **-oX** is used with **-bd**, or when **-q** with a time is used without **-bd**, this is the only way of causing Exim to write a pid file, because in those cases, the normal pid file is not used.

-or <*time*>
This option sets a timeout value for incoming non-SMTP messages. If it is not set, Exim will wait forever for the standard input. The value can also be set by the **receive_timeout** option. The format used for specifying times is described in section 6.15.

-os <*time*>
This option sets a timeout value for incoming SMTP messages. The timeout applies to each SMTP command and block of data. The value can also be set by the **smtp_receive_timeout** option; it defaults to 5 minutes. The format used for specifying times is described in section 6.15.

-ov
This option has exactly the same effect as **-v**.

-oX <*number or string*>
This option is relevant only when the **-bd** (start listening daemon) option is also given. It controls which ports and interfaces the daemon uses. Details of the syntax, and how it interacts with configuration file options, are given in chapter 13. When **-oX** is used to start a daemon, no pid file is written unless **-oP** is also present to specify a pid file name.

-pd
This option applies when an embedded Perl interpreter is linked with Exim (see chapter 12). It overrides the setting of the **perl_at_start** option, forcing the starting of the interpreter to be delayed until it is needed.

-ps
This option applies when an embedded Perl interpreter is linked with Exim (see chapter 12). It overrides the setting of the **perl_at_start** option, forcing the starting of the interpreter to occur as soon as Exim is started.

-p<*rval*>:<*sval*>
For compatibility with Sendmail, this option is equivalent to

```
-oMr <rval> -oMs <sval>
```

It sets the incoming protocol and host name (for trusted callers). The host name and its colon can be omitted when only the protocol is to be set. Note the Exim already has two private options, **-pd** and **-ps**, that refer to embedded Perl. It is therefore impossible to set a protocol value of `p` or `s` using this option (but that does not seem a real limitation).

-q
This option is normally restricted to admin users. However, there is a configuration option called **prod_requires_admin** which can be set false to relax this restriction (and also the same requirement for the **-M**, **-R**, and **-S** options).

The **-q** option starts one queue runner process. This scans the queue of waiting messages, and runs a delivery process for each one in turn. It waits for each delivery process to finish before starting the next one. A delivery process may not actually do any deliveries if the retry times for the addresses have not been reached. Use **-qf** (see below) if you want to override this.

If the delivery process spawns other processes to deliver other messages down passed SMTP connections, the queue runner waits for these to finish before proceeding.

When all the queued messages have been considered, the original queue runner process terminates. In other words, a single pass is made over the waiting mail, one message at a time. Use **-q** with a time (see below) if you want this to be repeated periodically.

Exim processes the waiting messages in an unpredictable order. It isn't very random, but it is likely to be different each time, which is all that matters. If one particular message screws up a remote MTA, other messages to the same MTA have a chance of getting through if they get tried first.

It is possible to cause the messages to be processed in lexical message id order, which is essentially the order in which they arrived, by setting the **queue_run_in_order** option, but this is not recommended for normal use.

-q<*qflags*>
The **-q** option may be followed by one or more flag letters that change its behaviour. They are all optional, but if more than one is present, they must appear in the correct order. Each flag is described in a separate item below.

-qq...
An option starting with **-qq** requests a two-stage queue run. In the first stage, the queue is scanned as if the **queue_smtp_domains** option matched every domain. Addresses are routed, local deliveries happen, but no remote transports are run.

The hints database that remembers which messages are waiting for specific hosts is updated, as if delivery to those hosts had been deferred. After this is complete, a second, normal queue scan happens, with routing and delivery taking place as normal. Messages that are routed to the same host should mostly be delivered down a single SMTP connection because of the hints that were set up during the first queue scan. This option may be useful for hosts that are connected to the Internet intermittently.

-q[q]i...
If the *i* flag is present, the queue runner runs delivery processes only for those messages that haven't previously been tried. (*i* stands for "initial delivery".) This can be helpful if you are putting messages on the queue using **-odq** and want a queue runner just to process the new messages.

-q[q][i]f...
If one *f* flag is present, a delivery attempt is forced for each non-frozen message, whereas without *f* only those non-frozen addresses that have passed their retry times are tried.

-q[q][i]ff...
If *ff* is present, a delivery attempt is forced for every message, whether frozen or not.

-q[q][i][f[f]]l
The *l* (the letter "ell") flag specifies that only local deliveries are to be done. If a message requires any remote deliveries, it remains on the queue for later delivery.

-q<*qflags*> <*start id*> <*end id*>
When scanning the queue, Exim can be made to skip over messages whose ids are lexically less than a given value by following the **-q** option with a starting message id. For example:

```
exim -q 0t5C6f-0000c8-00
```

Messages that arrived earlier than `0t5C6f-0000c8-00` are not inspected. If a second message id is given, messages whose ids are lexically greater than it are also skipped. If the same id is given twice, for example,

```
exim -q 0t5C6f-0000c8-00 0t5C6f-0000c8-00
```

just one delivery process is started, for that message. This differs from **-M** in that retry data is respected, and it also differs from **-Mc** in that it counts as a delivery from a queue run. Note that the selection mechanism does not affect the order in which the messages are scanned. There are also other ways of selecting specific sets of messages for delivery in a queue run – see **-R** and **-S**.

-q<*qflags*><*time*>
When a time value is present, the **-q** option causes Exim to run as a daemon, starting a queue runner process at intervals specified by the given time value (whose format is described in section 6.15). This form of the **-q** option is commonly combined with the **-bd** option, in which case a single daemon process handles both functions. A common way of starting up a combined daemon at system boot time is to use a command such as

```
/usr/exim/bin/exim -bd -q30m
```

Such a daemon listens for incoming SMTP calls, and also starts a queue runner process every 30 minutes.

When a daemon is started by **-q** with a time value, but without **-bd**, no pid file is written unless one is explicitly requested by the **-oP** option.

-qR<*rsflags*> <*string*>
This option is synonymous with **-R**. It is provided for Sendmail compatibility.

-qS<*rsflags*> <*string*>
This option is synonymous with **-S**.

-R<*rsflags*> <*string*>
The <*rsflags*> may be empty, in which case the white space before the string is optional, unless the string is *f*, *ff*, *r*, *rf*, or *rff*, which are the possible values for <*rsflags*>. White space is required if <*rsflags*> is not empty.

This option is similar to **-q** with no time value, that is, it causes Exim to perform a single queue run, except that, when scanning the messages on the queue, Exim processes only those that have at least one undelivered recipient address containing the given string, which is checked in a case-independent way. If the <*rsflags*> start with *r*, <*string*> is interpreted as a regular expression; otherwise it is a literal string.

If you want to do periodic queue runs for messages with specific recipients, you can combine **-R** with **-q** and a time value. For example:

```
exim -q25m -R @special.domain.example
```

This example does a queue run for messages with recipients in the given domain every 25 minutes. Any additional flags that are specified with **-q** are applied to each queue run.

Once a message is selected for delivery by this mechanism, all its addresses are processed. For the first selected message, Exim overrides any retry information and forces a delivery attempt for each undelivered address. This means that if delivery of any address in the first message is successful, any existing retry information is deleted, and so delivery attempts for that address in subsequently selected messages (which are processed without forcing) will run. However, if delivery of any address does not succeed, the retry information is updated, and in subsequently selected messages, the failing address will be skipped.

If the <*rsflags*> contain *f* or *ff*, the delivery forcing applies to all selected messages, not just the first; frozen messages are included when *ff* is present.

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. When the SMTP command ETRN is accepted by its ACL (see chapter 40), its default effect is to run Exim with the **-R** option, but it can be configured to run an arbitrary command instead.

-r
This is a documented (for Sendmail) obsolete alternative name for **-f**.

-S<*rsflags*> <*string*>
This option acts like **-R** except that it checks the string against each message's sender instead of against the recipients. If **-R** is also set, both conditions must be met for a message to be selected. If either of the options has *f* or *ff* in its flags, the associated action is taken.

-Tqt <*times*>
This an option that is exclusively for use by the Exim testing suite. It is not recognized when Exim is run normally. It allows for the setting up of explicit "queue times" so that various warning/retry features can be tested.

-t
When Exim is receiving a locally-generated, non-SMTP message on its standard input, the **-t** option causes the recipients of the message to be obtained from the *To:*, *Cc:*, and *Bcc:* header lines in the message instead of from the command arguments. The addresses are extracted before any rewriting takes place and the *Bcc:* header line, if present, is then removed.

If the command has any arguments, they specify addresses to which the message is *not* to be delivered. That is, the argument addresses are removed from the recipients list obtained from the headers. This is compatible with Smail 3 and in accordance with the documented behaviour of several versions of Sendmail, as described in man pages on a number of operating systems (e.g. Solaris 8, IRIX 6.5, HP-UX 11). However, some versions of Sendmail *add* argument addresses to those obtained from the headers, and the O'Reilly Sendmail book documents it that way. Exim can be made to add argument addresses instead of subtracting them by setting the option **extract_addresses_remove_arguments** false.

If there are any **Resent-** header lines in the message, Exim extracts recipients from all *Resent-To:*, *Resent-Cc:*, and *Resent-Bcc:* header lines instead of from *To:*, *Cc:*, and *Bcc:*. This is for compatibility with Sendmail and other MTAs. (Prior to release 4.20, Exim gave an error if **-t** was used in conjunction with **Resent-** header lines.)

RFC 2822 talks about different sets of **Resent-** header lines (for when a message is resent several times). The RFC also specifies that they should be added at the front of the message, and separated by *Received:* lines. It is not at all clear how **-t** should operate in the present of multiple sets, nor indeed exactly what constitutes a "set". In practice, it seems that MUAs do not follow the RFC. The **Resent-** lines are often added at the end of the header, and if a message is resent more than once, it is common for the original set of **Resent-** headers to be renamed as **X-Resent-** when a new set is added. This removes any possible ambiguity.

-ti
This option is exactly equivalent to **-t -i**. It is provided for compatibility with Sendmail.

-tls-on-connect
This option is available when Exim is compiled with TLS support. It forces all incoming SMTP connections to behave as if the incoming port is listed in the **tls_on_connect_ports** option. See section 13.4 and chapter 39 for further details.

-U
Sendmail uses this option for "initial message submission", and its documentation states that in future releases, it may complain about syntactically invalid messages rather than fixing them when this flag is not set. Exim ignores this option.

-v
This option causes Exim to write information to the standard error stream, describing what it is doing. In particular, it shows the log lines for receiving and delivering a message, and if an SMTP connection is made, the SMTP dialogue is shown. Some of the log lines shown may not actually be written to the log if the setting of **log_selector** discards them. Any relevant selectors are shown with each log line. If none are shown, the logging is unconditional.

-x
AIX uses **-x** for a private purpose ("mail from a local mail program has National Language Support extended characters in the body of the mail item"). It sets **-x** when calling the MTA from its **mail** command. Exim ignores this option.

6. The Exim run time configuration file

Exim uses a single run time configuration file that is read whenever an Exim binary is executed. Note that in normal operation, this happens frequently, because Exim is designed to operate in a distributed manner, without central control.

If a syntax error is detected while reading the configuration file, Exim writes a message on the standard error, and exits with a non-zero return code. The message is also written to the panic log. **Note**: Only simple syntax errors can be detected at this time. The values of any expanded options are not checked until the expansion happens, even when the expansion does not actually alter the string.

The name of the configuration file is compiled into the binary for security reasons, and is specified by the CONFIGURE_FILE compilation option. In most configurations, this specifies a single file. However, it is permitted to give a colon-separated list of file names, in which case Exim uses the first existing file in the list.

The run time configuration file must be owned by root or by the user that is specified at compile time by the EXIM_USER option, or by the user that is specified at compile time by the CONFIGURE_OWNER option (if set). The configuration file must not be world-writeable or group-writeable, unless its group is the one specified at compile time by the EXIM_GROUP option or by the CONFIGURE_GROUP option.

Warning: In a conventional configuration, where the Exim binary is setuid to root, anybody who is able to edit the run time configuration file has an easy way to run commands as root. If you make your mail administrators members of the Exim group, but do not trust them with root, make sure that the run time configuration is not group writeable.

A default configuration file, which will work correctly in simple situations, is provided in the file *src/configure.default*. If CONFIGURE_FILE defines just one file name, the installation process copies the default configuration to a new file of that name if it did not previously exist. If CONFIGURE_FILE is a list, no default is automatically installed. Chapter 7 is a "walk-through" discussion of the default configuration.

6.1 Using a different configuration file

A one-off alternate configuration can be specified by the **-C** command line option, which may specify a single file or a list of files. However, when **-C** is used, Exim gives up its root privilege, unless called by root or the Exim user (or unless the argument for **-C** is identical to the built-in value from CONFIGURE_FILE). **-C** is useful mainly for checking the syntax of configuration files before installing them. No owner or group checks are done on a configuration file specified by **-C**.

The privileged use of **-C** by the Exim user can be locked out by setting ALT_CONFIG_ROOT_ONLY in *Local/Makefile* when building Exim. However, if you do this, you also lock out the possibility of testing a configuration using **-C** right through message reception and delivery, even if the caller is root. The reception works, but by that time, Exim is running as the Exim user, so when it re-execs to regain privilege for the delivery, the use of **-C** causes privilege to be lost. However, root can test reception and delivery using two separate commands (one to put a message on the queue, using **-odq**, and another to do the delivery, using **-M**).

If ALT_CONFIG_PREFIX is defined *in Local/Makefile*, it specifies a prefix string with which any file named in a **-C** command line option must start. In addition, the file name must not contain the sequence "/ . . /". There is no default setting for ALT_CONFIG_PREFIX; when it is unset, any file name can be used with **-C**.

One-off changes to a configuration can be specified by the **-D** command line option, which defines and overrides values for macros used inside the configuration file. However, like **-C**, the use of this option by a non-privileged user causes Exim to discard its root privilege. If DISABLE_D_OPTION is defined in *Local/Makefile*, the use of **-D** is completely disabled, and its use causes an immediate error exit.

Some sites may wish to use the same Exim binary on different machines that share a file system, but to use different configuration files on each machine. If CONFIGURE_FILE_USE_NODE is defined

in *Local/Makefile*, Exim first looks for a file whose name is the configuration file name followed by a dot and the machine's node name, as obtained from the *uname()* function. If this file does not exist, the standard name is tried. This processing occurs for each file name in the list given by CONFIGURE_FILE or **-C**.

In some esoteric situations different versions of Exim may be run under different effective uids and the CONFIGURE_FILE_USE_EUID is defined to help with this. See the comments in *src/EDITME* for details.

6.2 Configuration file format

Exim's configuration file is divided into a number of different parts. General option settings must always appear at the start of the file. The other parts are all optional, and may appear in any order. Each part other than the first is introduced by the word "begin" followed by the name of the part. The optional parts are:

- *ACL*: Access control lists for controlling incoming SMTP mail (see chapter 40).
- *authenticators*: Configuration settings for the authenticator drivers. These are concerned with the SMTP AUTH command (see chapter 33).
- *routers*: Configuration settings for the router drivers. Routers process addresses and determine how the message is to be delivered (see chapters 15–22).
- *transports*: Configuration settings for the transport drivers. Transports define mechanisms for copying messages to destinations (see chapters 24–30).
- *retry*: Retry rules, for use when a message cannot be delivered immediately. If there is no retry section, or if it is empty (that is, no retry rules are defined), Exim will not retry deliveries. In this situation, temporary errors are treated the same as permanent errors. Retry rules are discussed in chapter 32.
- *rewrite*: Global address rewriting rules, for use when a message arrives and when new addresses are generated during delivery. Rewriting is discussed in chapter 31.
- *local_scan*: Private options for the *local_scan()* function. If you want to use this feature, you must set

  ```
  LOCAL_SCAN_HAS_OPTIONS=yes
  ```

  in *Local/Makefile* before building Exim. Details of the *local_scan()* facility are given in chapter 42.

Leading and trailing white space in configuration lines is always ignored.

Blank lines in the file, and lines starting with a # character (ignoring leading white space) are treated as comments and are ignored. **Note**: A # character other than at the beginning of a line is not treated specially, and does not introduce a comment.

Any non-comment line can be continued by ending it with a backslash. Note that the general rule for white space means that trailing white space after the backslash and leading white space at the start of continuation lines is ignored. Comment lines beginning with # (but not empty lines) may appear in the middle of a sequence of continuation lines.

A convenient way to create a configuration file is to start from the default, which is supplied in *src/configure.default*, and add, delete, or change settings as required.

The ACLs, retry rules, and rewriting rules have their own syntax which is described in chapters 40, 32, and 31, respectively. The other parts of the configuration file have some syntactic items in common, and these are described below, from section 6.10 onwards. Before that, the inclusion, macro, and conditional facilities are described.

6.3 File inclusions in the configuration file

You can include other files inside Exim's run time configuration file by using this syntax:

```
.include <file name>
.include_if_exists <file name>
```

on a line by itself. Double quotes round the file name are optional. If you use the first form, a configuration error occurs if the file does not exist; the second form does nothing for non-existent files. In all cases, an absolute file name is required.

Includes may be nested to any depth, but remember that Exim reads its configuration file often, so it is a good idea to keep them to a minimum. If you change the contents of an included file, you must HUP the daemon, because an included file is read only when the configuration itself is read.

The processing of inclusions happens early, at a physical line level, so, like comment lines, an inclusion can be used in the middle of an option setting, for example:

```
hosts_lookup = a.b.c \
               .include /some/file
```

Include processing happens after macro processing (see below). Its effect is to process the lines of the included file as if they occurred inline where the inclusion appears.

6.4 Macros in the configuration file

If a line in the main part of the configuration (that is, before the first "begin" line) begins with an upper case letter, it is taken as a macro definition, and must be of the form

<name> = <rest of line>

The name must consist of letters, digits, and underscores, and need not all be in upper case, though that is recommended. The rest of the line, including any continuations, is the replacement text, and has leading and trailing white space removed. Quotes are not removed. The replacement text can never end with a backslash character, but this doesn't seem to be a serious limitation.

Macros may also be defined between router, transport, authenticator, or ACL definitions. They may not, however, be defined within an individual driver or ACL, or in the **local_scan**, retry, or rewrite sections of the configuration.

6.5 Macro substitution

Once a macro is defined, all subsequent lines in the file (and any included files) are scanned for the macro name; if there are several macros, the line is scanned for each in turn, in the order in which the macros are defined. The replacement text is not re-scanned for the current macro, though it is scanned 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

```
ABCD_XYZ = <something>
ABCD = <something else>
```

but putting the definitions in the opposite order would provoke a configuration error. Macro expansion is applied to individual physical lines from the file, before checking for line continuation or file inclusion (see above). If a line consists solely of a macro name, and the expansion of the macro is empty, the line is ignored. A macro at the start of a line may turn the line into a comment line or a `.include` line.

6.6 Redefining macros

Once defined, the value of a macro can be redefined later in the configuration (or in an included file). Redefinition is specified by using == instead of =. For example:

```
MAC =  initial value
...
MAC == updated value
```

Redefinition does not alter the order in which the macros are applied to the subsequent lines of the configuration file. It is still the same order in which the macros were originally defined. All that changes is the macro's value. Redefinition makes it possible to accumulate values. For example:

```
MAC =  initial value
...
MAC == MAC and something added
```

This can be helpful in situations where the configuration file is built from a number of other files.

6.7 Overriding macro values

The values set for macros in the configuration file can be overridden by the **-D** command line option, but Exim gives up its root privilege when **-D** is used, unless called by root or the Exim user. A definition on the command line using the **-D** option causes all definitions and redefinitions within the file to be ignored.

6.8 Example of macro usage

As an example of macro usage, consider a configuration where aliases are looked up in a MySQL database. It helps to keep the file less cluttered if long strings such as SQL statements are defined separately as macros, for example:

```
ALIAS_QUERY = select mailbox from user where \
              login=${quote_mysql:$local_part};
```

This can then be used in a *redirect* router setting like this:

```
data = ${lookup mysql{ALIAS_QUERY}}
```

In earlier versions of Exim macros were sometimes used for domain, host, or address lists. In Exim 4 these are handled better by named lists – see section 10.5.

6.9 Conditional skips in the configuration file

You can use the directives `.ifdef`, `.ifndef`, `.elifdef`, `.elifndef`, `.else`, and `.endif` to dynamically include or exclude portions of the configuration file. The processing happens whenever the file is read (that is, when an Exim binary starts to run).

The implementation is very simple. Instances of the first four directives must be followed by text that includes the names of one or macros. The condition that is tested is whether or not any macro substitution has taken place in the line. Thus:

```
.ifdef AAA
message_size_limit = 50M
.else
message_size_limit = 100M
.endif
```

sets a message size limit of 50M if the macro `AAA` is defined, and 100M otherwise. If there is more than one macro named on the line, the condition is true if any of them are defined. That is, it is an "or" condition. To obtain an "and" condition, you need to use nested `.ifdef`s.

Although you can use a macro expansion to generate one of these directives, it is not very useful, because the condition "there was a macro substitution in this line" will always be true.

Text following `.else` and `.endif` is ignored, and can be used as comment to clarify complicated nestings.

6.10 Common option syntax

For the main set of options, driver options, and *local_scan()* options, each setting is on a line by itself, and starts with a name consisting of lower-case letters and underscores. Many options require a data

value, and in these cases the name must be followed by an equals sign (with optional white space) and then the value. For example:

```
qualify_domain = mydomain.example.com
```

Some option settings may contain sensitive data, for example, passwords for accessing databases. To stop non-admin users from using the **-bP** command line option to read these values, you can precede the option settings with the word "hide". For example:

```
hide mysql_servers = localhost/users/admin/secret-password
```

For non-admin users, such options are displayed like this:

```
mysql_servers = <value not displayable>
```

If "hide" is used on a driver option, it hides the value of that option on all instances of the same driver.

The following sections describe the syntax used for the different data types that are found in option settings.

6.11 Boolean options

Options whose type is given as boolean are on/off switches. There are two different ways of specifying such options: with and without a data value. If the option name is specified on its own without data, the switch is turned on; if it is preceded by "no_" or "not_" the switch is turned off. However, boolean options may be followed by an equals sign and one of the words "true", "false", "yes", or "no", as an alternative syntax. For example, the following two settings have exactly the same effect:

```
queue_only
queue_only = true
```

The following two lines also have the same (opposite) effect:

```
no_queue_only
queue_only = false
```

You can use whichever syntax you prefer.

6.12 Integer values

If an option's type is given as "integer", the value can be given in decimal, hexadecimal, or octal. If it starts with a digit greater than zero, a decimal number is assumed. Otherwise, it is treated as an octal number unless it starts with the characters "0x", in which case the remainder is interpreted as a hexadecimal number.

If an integer value is followed by the letter K, it is multiplied by 1024; if it is followed by the letter M, it is multiplied by 1024x1024. When the values of integer option settings are output, values which are an exact multiple of 1024 or 1024x1024 are sometimes, but not always, printed using the letters K and M. The printing style is independent of the actual input format that was used.

6.13 Octal integer values

If an option's type is given as "octal integer", its value is always interpreted as an octal number, whether or not it starts with the digit zero. Such options are always output in octal.

6.14 Fixed point numbers

If an option's type is given as "fixed-point", its value must be a decimal integer, optionally followed by a decimal point and up to three further digits.

6.15 Time intervals

A time interval is specified as a sequence of numbers, each followed by one of the following letters, with no intervening white space:

s seconds
m minutes
h hours
d days
w weeks

For example, “3h50m” specifies 3 hours and 50 minutes. The values of time intervals are output in the same format. Exim does not restrict the values; it is perfectly acceptable, for example, to specify “90m” instead of “1h30m”.

6.16 String values

If an option’s type is specified as “string”, the value can be specified with or without double-quotes. If it does not start with a double-quote, the value consists of the remainder of the line plus any continuation lines, starting at the first character after any leading white space, with trailing white space removed, and with no interpretation of the characters in the string. Because Exim removes comment lines (those beginning with #) at an early stage, they can appear in the middle of a multi-line string. The following two settings are therefore equivalent:

```
trusted_users = uucp:mail
trusted_users = uucp:\
                # This comment line is ignored
                mail
```

If a string does start with a double-quote, it must end with a closing double-quote, and any backslash characters other than those used for line continuation are interpreted as escape characters, as follows:

`\\`	single backslash
`\n`	newline
`\r`	carriage return
`\t`	tab
`\<`*octal digits*`>`	up to 3 octal digits specify one character
`\x<`*hex digits*`>`	up to 2 hexadecimal digits specify one character

If a backslash is followed by some other character, including a double-quote character, that character replaces the pair.

Quoting is necessary only if you want to make use of the backslash escapes to insert special characters, or if you need to specify a value with leading or trailing spaces. These cases are rare, so quoting is almost never needed in current versions of Exim. In versions of Exim before 3.14, quoting was required in order to continue lines, so you may come across older configuration files and examples that apparently quote unnecessarily.

6.17 Expanded strings

Some strings in the configuration file are subjected to *string expansion*, by which means various parts of the string may be changed according to the circumstances (see chapter 11). The input syntax for such strings is as just described; in particular, the handling of backslashes in quoted strings is done as part of the input process, before expansion takes place. However, backslash is also an escape character for the expander, so any backslashes that are required for that reason must be doubled if they are within a quoted configuration string.

6.18 User and group names

User and group names are specified as strings, using the syntax described above, but the strings are interpreted specially. A user or group name must either consist entirely of digits, or be a name that can be looked up using the *getpwnam()* or *getgrnam()* function, as appropriate.

6.19 List construction

The data for some configuration options is a list of items, with colon as the default separator. Many of these options are shown with type “string list” in the descriptions later in this document. Others are listed as “domain list”, “host list”, “address list”, or “local part list”. Syntactically, they are all the same; however, those other than “string list” are subject to particular kinds of interpretation, as described in chapter 10.

In all these cases, the entire list is treated as a single string as far as the input syntax is concerned. The **trusted_users** setting in section 6.16 above is an example. If a colon is actually needed in an item in a list, it must be entered as two colons. Leading and trailing white space on each item in a 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, the list

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

contains two IP addresses, the IPv4 address 127.0.0.1 and the IPv6 address ::1.

Note: Although leading and trailing white space is ignored in individual list items, it is not ignored when parsing the list. The space after the first colon in the example above is necessary. If it were not there, the list would be interpreted as the two items 127.0.0.1:: and 1.

6.20 Changing list separators

Doubling colons in IPv6 addresses is an unwelcome chore, so a mechanism was introduced to allow the separator character to be changed. If a list begins with a left angle bracket, followed by any punctuation character, that character is used instead of colon as the list separator. For example, the list above can be rewritten to use a semicolon separator like this:

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

This facility applies to all lists, with the exception of the list in **log_file_path**. It is recommended that the use of non-colon separators be confined to circumstances where they really are needed.

It is also possible to use newline and other control characters (those with code values less than 32, plus DEL) as separators in lists. Such separators must be provided literally at the time the list is processed. For options that are string-expanded, you can write the separator using a normal escape sequence. This will be processed by the expander before the string is interpreted as a list. For example, if a newline-separated list of domains is generated by a lookup, you can process it directly by a line such as this:

```
domains = <\n ${lookup mysql{.....}}
```

This avoids having to change the list separator in such data. You are unlikely to want to use a control character as a separator in an option that is not expanded, because the value is literal text. However, it can be done by giving the value in quotes. For example:

```
local_interfaces = "<\n 127.0.0.1 \n ::1"
```

Unlike printing character separators, which can be included in list items by doubling, it is not possible to include a control character as data when it is set as the separator. Two such characters in succession are interpreted as enclosing an empty list item.

6.21 Empty items in lists

An empty item at the end of a list is always ignored. In other words, trailing separator characters are ignored. Thus, the list in

```
senders = user@domain :
```

contains only a single item. If you want to include an empty string as one item in a list, it must not be the last item. For example, this list contains three items, the second of which is empty:

```
senders = user1@domain : : user2@domain
```

Note: There must be white space between the two colons, as otherwise they are interpreted as representing a single colon data character (and the list would then contain just one item). If you want to specify a list that contains just one, empty item, you can do it as in this example:

```
senders = :
```

In this case, the first item is empty, and the second is discarded because it is at the end of the list.

6.22 Format of driver configurations

There are separate parts in the configuration for defining routers, transports, and authenticators. In each part, you are defining a number of driver instances, each with its own set of options. Each driver instance is defined by a sequence of lines like this:

<*instance name*>:
 <*option*>
 ...
 <*option*>

In the following example, the instance name is *localuser*, and it is followed by three options settings:

```
localuser:
  driver = accept
  check_local_user
  transport = local_delivery
```

For each driver instance, you specify which Exim code module it uses – by the setting of the **driver** option – and (optionally) some configuration settings. For example, in the case of transports, if you want a transport to deliver with SMTP you would use the *smtp* driver; if you want to deliver to a local file you would use the *appendfile* driver. Each of the drivers is described in detail in its own separate chapter later in this manual.

You can have several routers, transports, or authenticators that are based on the same underlying driver (each must have a different instance name).

The order in which routers are defined is important, because addresses are passed to individual routers one by one, in order. The order in which transports are defined does not matter at all. The order in which authenticators are defined is used only when Exim, as a client, is searching them to find one that matches an authentication mechanism offered by the server.

Within a driver instance definition, there are two kinds of option: *generic* and *private*. The generic options are those that apply to all drivers of the same type (that is, all routers, all transports or all authenticators). The **driver** option is a generic option that must appear in every definition. The private options are special for each driver, and none need appear, because they all have default values.

The options may appear in any order, except that the **driver** option must precede any private options, since these depend on the particular driver. For this reason, it is recommended that **driver** always be the first option.

Driver instance names, which are used for reference in log entries and elsewhere, can be any sequence of letters, digits, and underscores (starting with a letter) and must be unique among drivers of the same type. A router and a transport (for example) can each have the same name, but no two router instances can have the same name. The name of a driver instance should not be confused with the name of the underlying driver module. For example, the configuration lines:

```
remote_smtp:
  driver = smtp
```

create an instance of the *smtp* transport driver whose name is *remote_smtp*. The same driver code can be used more than once, with different instance names and different option settings each time. A second instance of the *smtp* transport, with different options, might be defined thus:

```
special_smtp:
  driver = smtp
```

```
port = 1234
command_timeout = 10s
```

The names *remote_smtp* and *special_smtp* would be used to reference these transport instances from routers, and these names would appear in log lines.

Comment lines may be present in the middle of driver specifications. The full list of option settings for any particular driver instance, including all the defaulted values, can be extracted by making use of the **-bP** command line option.

7. The default configuration file

The default configuration file supplied with Exim as *src/configure.default* is sufficient for a host with simple mail requirements. As an introduction to the way Exim is configured, this chapter "walks through" the default configuration, giving brief explanations of the settings. Detailed descriptions of the options are given in subsequent chapters. The default configuration file itself contains extensive comments about ways you might want to modify the initial settings. However, note that there are many options that are not mentioned at all in the default configuration.

7.1 Main configuration settings

The main (global) configuration option settings must always come first in the file. The first thing you'll see in the file, after some initial comments, is the line

```
# primary_hostname =
```

This is a commented-out setting of the **primary_hostname** option. Exim needs to know the official, fully qualified name of your host, and this is where you can specify it. However, in most cases you do not need to set this option. When it is unset, Exim uses the *uname()* system function to obtain the host name.

The first three non-comment configuration lines are as follows:

```
domainlist local_domains = @
domainlist relay_to_domains =
hostlist   relay_from_hosts = 127.0.0.1
```

These are not, in fact, option settings. They are definitions of two named domain lists and one named host list. Exim allows you to give names to lists of domains, hosts, and email addresses, in order to make it easier to manage the configuration file (see section 10.5).

The first line defines a domain list called *local_domains*; this is used later in the configuration to identify domains that are to be delivered on the local host.

There is just one item in this list, the string "@". This is a special form of entry which means "the name of the local host". Thus, if the local host is called *a.host.example*, mail to *any.user@a.host.example* is expected to be delivered locally. Because the local host's name is referenced indirectly, the same configuration file can be used on different hosts.

The second line defines a domain list called *relay_to_domains*, but the list itself is empty. Later in the configuration we will come to the part that controls mail relaying through the local host; it allows relaying to any domains in this list. By default, therefore, no relaying on the basis of a mail domain is permitted.

The third line defines a host list called *relay_from_hosts*. This list is used later in the configuration to permit relaying from any host or IP address that matches the list. The default contains just the IP address of the IPv4 loopback interface, which means that processes on the local host are able to submit mail for relaying by sending it over TCP/IP to that interface. No other hosts are permitted to submit messages for relaying.

Just to be sure there's no misunderstanding: at this point in the configuration we aren't actually setting up any controls. We are just defining some domains and hosts that will be used in the controls that are specified later.

The next two configuration lines are genuine option settings:

```
acl_smtp_rcpt = acl_check_rcpt
acl_smtp_data = acl_check_data
```

These options specify *Access Control Lists* (ACLs) that are to be used during an incoming SMTP session for every recipient of a message (every RCPT command), and after the contents of the message have been received, respectively. The names of the lists are *acl_check_rcpt* and *acl_check_data*, and we will come to their definitions below, in the ACL section of the configuration. The RCPT ACL controls which recipients are accepted for an incoming message – if a configuration

does not provide an ACL to check recipients, no SMTP mail can be accepted. The DATA ACL allows the contents of a message to be checked.

Two commented-out option settings are next:

```
# av_scanner = clamd:/tmp/clamd
# spamd_address = 127.0.0.1 783
```

These are example settings that can be used when Exim is compiled with the content-scanning extension. The first specifies the interface to the virus scanner, and the second specifies the interface to SpamAssassin. Further details are given in chapter 41.

Three more commented-out option settings follow:

```
# tls_advertise_hosts = *
# tls_certificate = /etc/ssl/exim.crt
# tls_privatekey = /etc/ssl/exim.pem
```

These are example settings that can be used when Exim is compiled with support for TLS (aka SSL) as described in section 4.6. The first one specifies the list of clients that are allowed to use TLS when connecting to this server; in this case the wildcard means all clients. The other options specify where Exim should find its TLS certificate and private key, which together prove the server's identity to any clients that connect. More details are given in chapter 39.

Another two commented-out option settings follow:

```
# daemon_smtp_ports = 25 : 465 : 587
# tls_on_connect_ports = 465
```

These options provide better support for roaming users who wish to use this server for message submission. They are not much use unless you have turned on TLS (as described in the previous paragraph) and authentication (about which more in section 7.7). The usual SMTP port 25 is often blocked on end-user networks, so RFC 4409 specifies that message submission should use port 587 instead. However some software (notably Microsoft Outlook) cannot be configured to use port 587 correctly, so these settings also enable the non-standard "smtps" (aka "ssmtp") port 465 (see section 13.4).

Two more commented-out options settings follow:

```
# qualify_domain =
# qualify_recipient =
```

The first of these specifies a domain that Exim uses when it constructs a complete email address from a local login name. This is often needed when Exim receives a message from a local process. If you do not set **qualify_domain**, the value of **primary_hostname** is used. If you set both of these options, you can have different qualification domains for sender and recipient addresses. If you set only the first one, its value is used in both cases.

The following line must be uncommented if you want Exim to recognize addresses of the form *user@[10.11.12.13]* that is, with a "domain literal" (an IP address within square brackets) instead of a named domain.

```
# allow_domain_literals
```

The RFCs still require this form, but many people think that in the modern Internet it makes little sense to permit mail to be sent to specific hosts by quoting their IP addresses. This ancient format has been used by people who try to abuse hosts by using them for unwanted relaying. However, some people believe there are circumstances (for example, messages addressed to *postmaster*) where domain literals are still useful.

The next configuration line is a kind of trigger guard:

```
never_users = root
```

It specifies that no delivery must ever be run as the root user. The normal convention is to set up *root* as an alias for the system administrator. This setting is a guard against slips in the configuration. The list of users specified by **never_users** is not, however, the complete list; the build-time configuration

in *Local/Makefile* has an option called FIXED_NEVER_USERS specifying a list that cannot be overridden. The contents of **never_users** are added to this list. By default FIXED_NEVER_USERS also specifies root.

When a remote host connects to Exim in order to send mail, the only information Exim has about the host's identity is its IP address. The next configuration line,

```
host_lookup = *
```

specifies that Exim should do a reverse DNS lookup on all incoming connections, in order to get a host name. This improves the quality of the logging information, but if you feel it is too expensive, you can remove it entirely, or restrict the lookup to hosts on "nearby" networks. Note that it is not always possible to find a host name from an IP address, because not all DNS reverse zones are maintained, and sometimes DNS servers are unreachable.

The next two lines are concerned with *ident* callbacks, as defined by RFC 1413 (hence their names):

```
rfc1413_hosts = *
rfc1413_query_timeout = 5s
```

These settings cause Exim to make ident callbacks for all incoming SMTP calls. You can limit the hosts to which these calls are made, or change the timeout that is used. If you set the timeout to zero, all ident calls are disabled. Although they are cheap and can provide useful information for tracing problem messages, some hosts and firewalls have problems with ident calls. This can result in a timeout instead of an immediate refused connection, leading to delays on starting up an incoming SMTP session.

When Exim receives messages over SMTP connections, it expects all addresses to be fully qualified with a domain, as required by the SMTP definition. However, if you are running a server to which simple clients submit messages, you may find that they send unqualified addresses. The two commented-out options:

```
# sender_unqualified_hosts =
# recipient_unqualified_hosts =
```

show how you can specify hosts that are permitted to send unqualified sender and recipient addresses, respectively.

The **percent_hack_domains** option is also commented out:

```
# percent_hack_domains =
```

It provides a list of domains for which the "percent hack" is to operate. This is an almost obsolete form of explicit email routing. If you do not know anything about it, you can safely ignore this topic.

The last two settings in the main part of the default configuration are concerned with messages that have been "frozen" on Exim's queue. When a message is frozen, Exim no longer continues to try to deliver it. Freezing occurs when a bounce message encounters a permanent failure because the sender address of the original message that caused the bounce is invalid, so the bounce cannot be delivered. This is probably the most common case, but there are also other conditions that cause freezing, and frozen messages are not always bounce messages.

```
ignore_bounce_errors_after = 2d
timeout_frozen_after = 7d
```

The first of these options specifies that failing bounce messages are to be discarded after 2 days on the queue. The second specifies that any frozen message (whether a bounce message or not) is to be timed out (and discarded) after a week. In this configuration, the first setting ensures that no failing bounce message ever lasts a week.

7.2 ACL configuration

In the default configuration, the ACL section follows the main configuration. It starts with the line

```
begin acl
```

and it contains the definitions of two ACLs, called *acl_check_rcpt* and *acl_check_data*, that were referenced in the settings of **acl_smtp_rcpt** and **acl_smtp_data** above.

The first ACL is used for every RCPT command in an incoming SMTP message. Each RCPT command specifies one of the message's recipients. The ACL statements are considered in order, until the recipient address is either accepted or rejected. The RCPT command is then accepted or rejected, according to the result of the ACL processing.

```
acl_check_rcpt:
```

This line, consisting of a name terminated by a colon, marks the start of the ACL, and names it.

```
accept  hosts = :
```

This ACL statement accepts the recipient if the sending host matches the list. But what does that strange list mean? It doesn't actually contain any host names or IP addresses. The presence of the colon puts an empty item in the list; Exim matches this only if the incoming message did not come from a remote host, because in that case, the remote hostname is empty. The colon is important. Without it, the list itself is empty, and can never match anything.

What this statement is doing is to accept unconditionally all recipients in messages that are submitted by SMTP from local processes using the standard input and output (that is, not using TCP/IP). A number of MUAs operate in this manner.

```
deny    message       = Restricted characters in address
        domains       = +local_domains
        local_parts   = ^[.] : ^.*[@%!/|]

deny    message       = Restricted characters in address
        domains       = !+local_domains
        local_parts   = ^[./|] : ^.*[@%!] : ^.*/\\.\\./
```

These statements are concerned with local parts that contain any of the characters "@", "%", "!", "/", "|", or dots in unusual places. Although these characters are entirely legal in local parts (in the case of "@" and leading dots, only if correctly quoted), they do not commonly occur in Internet mail addresses.

The first three have in the past been associated with explicitly routed addresses (percent is still sometimes used – see the **percent_hack_domains** option). Addresses containing these characters are regularly tried by spammers in an attempt to bypass relaying restrictions, and also by open relay testing programs. Unless you really need them it is safest to reject these characters at this early stage. This configuration is heavy-handed in rejecting these characters for all messages it accepts from remote hosts. This is a deliberate policy of being as safe as possible.

The first rule above is stricter, and is applied to messages that are addressed to one of the local domains handled by this host. This is implemented by the first condition, which restricts it to domains that are listed in the *local_domains* domain list. The "+" character is used to indicate a reference to a named list. In this configuration, there is just one domain in *local_domains*, but in general there may be many.

The second condition on the first statement uses two regular expressions to block local parts that begin with a dot or contain "@", "%", "!", "/", or "|". If you have local accounts that include these characters, you will have to modify this rule.

Empty components (two dots in a row) are not valid in RFC 2822, but Exim allows them because they have been encountered in practice. (Consider the common convention of local parts constructed as "*first-initial.second-initial.family-name*" when applied to someone like the author of Exim, who has no second initial.) However, a local part starting with a dot or containing "/../" can cause trouble if it is used as part of a file name (for example, for a mailing list). This is also true for local parts that contain slashes. A pipe symbol can also be troublesome if the local part is incorporated unthinkingly into a shell command line.

The second rule above applies to all other domains, and is less strict. This allows your own users to send outgoing messages to sites that use slashes and vertical bars in their local parts. It blocks local

parts that begin with a dot, slash, or vertical bar, but allows these characters within the local part. However, the sequence "/../" is barred. The use of "@", "%", and "!" is blocked, as before. The motivation here is to prevent your users (or your users' viruses) from mounting certain kinds of attack on remote sites.

```
accept  local_parts   = postmaster
        domains       = +local_domains
```

This statement, which has two conditions, accepts an incoming address if the local part is *postmaster* and the domain is one of those listed in the *local_domains* domain list. The "+" character is used to indicate a reference to a named list. In this configuration, there is just one domain in *local_domains*, but in general there may be many.

The presence of this statement means that mail to postmaster is never blocked by any of the subsequent tests. This can be helpful while sorting out problems in cases where the subsequent tests are incorrectly denying access.

```
require verify        = sender
```

This statement requires the sender address to be verified before any subsequent ACL statement can be used. If verification fails, the incoming recipient address is refused. Verification consists of trying to route the address, to see if a bounce message could be delivered to it. In the case of remote addresses, basic verification checks only the domain, but *callouts* can be used for more verification if required. Section 40.40 discusses the details of address verification.

```
accept  hosts         = +relay_from_hosts
        control       = submission
```

This statement accepts the address if the message is coming from one of the hosts that are defined as being allowed to relay through this host. Recipient verification is omitted here, because in many cases the clients are dumb MUAs that do not cope well with SMTP error responses. For the same reason, the second line specifies "submission mode" for messages that are accepted. This is described in detail in section 44.1; it causes Exim to fix messages that are deficient in some way, for example, because they lack a *Date:* header line. If you are actually relaying out from MTAs, you should probably add recipient verification here, and disable submission mode.

```
accept  authenticated = *
        control       = submission
```

This statement accepts the address if the client host has authenticated itself. Submission mode is again specified, on the grounds that such messages are most likely to come from MUAs. The default configuration does not define any authenticators, though it does include some nearly complete commented-out examples described in 7.7. This means that no client can in fact authenticate until you complete the authenticator definitions.

```
require message = relay not permitted
        domains = +local_domains : +relay_domains
```

This statement rejects the address if its domain is neither a local domain nor one of the domains for which this host is a relay.

```
require verify = recipient
```

This statement requires the recipient address to be verified; if verification fails, the address is rejected.

```
# deny    message     = rejected because $sender_host_address \
#                       is in a black list at $dnslist_domain\n\
#                       $dnslist_text
#         dnslists    = black.list.example
#
# warn    dnslists    = black.list.example
#         add_header  = X-Warning: $sender_host_address is in \
#                       a black list at $dnslist_domain
#         log_message = found in $dnslist_domain
```

These commented-out lines are examples of how you could configure Exim to check sending hosts against a DNS black list. The first statement rejects messages from blacklisted hosts, whereas the second just inserts a warning header line.

```
# require verify = csa
```

This commented-out line is an example of how you could turn on client SMTP authorization (CSA) checking. Such checks do DNS lookups for special SRV records.

```
accept
```

The final statement in the first ACL unconditionally accepts any recipient address that has successfully passed all the previous tests.

```
acl_check_data:
```

This line marks the start of the second ACL, and names it. Most of the contents of this ACL are commented out:

```
# deny    malware   = *
#         message   = This message contains a virus \
#                     ($malware_name).
```

These lines are examples of how to arrange for messages to be scanned for viruses when Exim has been compiled with the content-scanning extension, and a suitable virus scanner is installed. If the message is found to contain a virus, it is rejected with the given custom error message.

```
# warn    spam      = nobody
#         message   = X-Spam_score: $spam_score\n\
#                     X-Spam_score_int: $spam_score_int\n\
#                     X-Spam_bar: $spam_bar\n\
#                     X-Spam_report: $spam_report
```

These lines are an example of how to arrange for messages to be scanned by SpamAssassin when Exim has been compiled with the content-scanning extension, and SpamAssassin has been installed. The SpamAssassin check is run with `nobody` as its user parameter, and the results are added to the message as a series of extra header line. In this case, the message is not rejected, whatever the spam score.

```
accept
```

This final line in the DATA ACL accepts the message unconditionally.

7.3 Router configuration

The router configuration comes next in the default configuration, introduced by the line

```
begin routers
```

Routers are the modules in Exim that make decisions about where to send messages. An address is passed to each router in turn, until it is either accepted, or failed. This means that the order in which you define the routers matters. Each router is fully described in its own chapter later in this manual. Here we give only brief overviews.

```
# domain_literal:
#   driver = ipliteral
#   domains = !+local_domains
#   transport = remote_smtp
```

This router is commented out because the majority of sites do not want to support domain literal addresses (those of the form *user@[10.9.8.7]*). If you uncomment this router, you also need to uncomment the setting of **allow_domain_literals** in the main part of the configuration.

```
dnslookup:
  driver = dnslookup
  domains = ! +local_domains
```

```
  transport = remote_smtp
  ignore_target_hosts = 0.0.0.0 : 127.0.0.0/8
  no_more
```

The first uncommented router handles addresses that do not involve any local domains. This is specified by the line

```
domains = ! +local_domains
```

The **domains** option lists the domains to which this router applies, but the exclamation mark is a negation sign, so the router is used only for domains that are not in the domain list called *local_domains* (which was defined at the start of the configuration). The plus sign before *local_domains* indicates that it is referring to a named list. Addresses in other domains are passed on to the following routers.

The name of the router driver is *dnslookup*, and is specified by the **driver** option. Do not be confused by the fact that the name of this router instance is the same as the name of the driver. The instance name is arbitrary, but the name set in the **driver** option must be one of the driver modules that is in the Exim binary.

The *dnslookup* router routes addresses by looking up their domains in the DNS in order to obtain a list of hosts to which the address is routed. If the router succeeds, the address is queued for the *remote_smtp* transport, as specified by the **transport** option. If the router does not find the domain in the DNS, no further routers are tried because of the **no_more** setting, so the address fails and is bounced.

The **ignore_target_hosts** option specifies a list of IP addresses that are to be entirely ignored. This option is present because a number of cases have been encountered where MX records in the DNS point to host names whose IP addresses are 0.0.0.0 or are in the 127 subnet (typically 127.0.0.1). Completely ignoring these IP addresses causes Exim to fail to route the email address, so it bounces. Otherwise, Exim would log a routing problem, and continue to try to deliver the message periodically until the address timed out.

```
system_aliases:
  driver = redirect
  allow_fail
  allow_defer
  data = ${lookup{$local_part}lsearch{/etc/aliases}}
# user = exim
  file_transport = address_file
  pipe_transport = address_pipe
```

Control reaches this and subsequent routers only for addresses in the local domains. This router checks to see whether the local part is defined as an alias in the */etc/aliases* file, and if so, redirects it according to the data that it looks up from that file. If no data is found for the local part, the value of the **data** option is empty, causing the address to be passed to the next router.

/etc/aliases is a conventional name for the system aliases file that is often used. That is why it is referenced by from the default configuration file. However, you can change this by setting SYSTEM_ ALIASES_FILE in *Local/Makefile* before building Exim.

```
userforward:
  driver = redirect
  check_local_user
# local_part_suffix = +* : -*
# local_part_suffix_optional
  file = $home/.forward
# allow_filter
  no_verify
  no_expn
  check_ancestor
  file_transport = address_file
```

```
  pipe_transport = address_pipe
  reply_transport = address_reply
```

This is the most complicated router in the default configuration. It is another redirection router, but this time it is looking for forwarding data set up by individual users. The **check_local_user** setting specifies a check that the local part of the address is the login name of a local user. If it is not, the router is skipped. The two commented options that follow **check_local_user**, namely:

```
# local_part_suffix = +* : -*
# local_part_suffix_optional
```

show how you can specify the recognition of local part suffixes. If the first is uncommented, a suffix beginning with either a plus or a minus sign, followed by any sequence of characters, is removed from the local part and placed in the variable *$local_part_suffix*. The second suffix option specifies that the presence of a suffix in the local part is optional. When a suffix is present, the check for a local login uses the local part with the suffix removed.

When a local user account is found, the file called *.forward* in the user's home directory is consulted. If it does not exist, or is empty, the router declines. Otherwise, the contents of *.forward* are interpreted as redirection data (see chapter 22 for more details).

Traditional *.forward* files contain just a list of addresses, pipes, or files. Exim supports this by default. However, if **allow_filter** is set (it is commented out by default), the contents of the file are interpreted as a set of Exim or Sieve filtering instructions, provided the file begins with "#Exim filter" or "#Sieve filter", respectively. User filtering is discussed in the separate document entitled *Exim's interfaces to mail filtering*.

The **no_verify** and **no_expn** options mean that this router is skipped when verifying addresses, or when running as a consequence of an SMTP EXPN command. There are two reasons for doing this:

(1) Whether or not a local user has a *.forward* file is not really relevant when checking an address for validity; it makes sense not to waste resources doing unnecessary work.

(2) More importantly, when Exim is verifying addresses or handling an EXPN command during an SMTP session, it is running as the Exim user, not as root. The group is the Exim group, and no additional groups are set up. It may therefore not be possible for Exim to read users' *.forward* files at this time.

The setting of **check_ancestor** prevents the router from generating a new address that is the same as any previous address that was redirected. (This works round a problem concerning a bad interaction between aliasing and forwarding – see section 22.5).

The final three option settings specify the transports that are to be used when forwarding generates a direct delivery to a file, or to a pipe, or sets up an auto-reply, respectively. For example, if a *.forward* file contains

```
a.nother@elsewhere.example, /home/spqr/archive
```

the delivery to */home/spqr/archive* is done by running the **address_file** transport.

```
localuser:
  driver = accept
  check_local_user
# local_part_suffix = +* : -*
# local_part_suffix_optional
  transport = local_delivery
```

The final router sets up delivery into local mailboxes, provided that the local part is the name of a local login, by accepting the address and assigning it to the *local_delivery* transport. Otherwise, we have reached the end of the routers, so the address is bounced. The commented suffix settings fulfil the same purpose as they do for the *userforward* router.

7.4 Transport configuration

Transports define mechanisms for actually delivering messages. They operate only when referenced from routers, so the order in which they are defined does not matter. The transports section of the configuration starts with

```
begin transports
```

One remote transport and four local transports are defined.

```
remote_smtp:
  driver = smtp
```

This transport is used for delivering messages over SMTP connections. All its options are defaulted. The list of remote hosts comes from the router.

```
local_delivery:
  driver = appendfile
  file = /var/mail/$local_part
  delivery_date_add
  envelope_to_add
  return_path_add
# group = mail
# mode = 0660
```

This *appendfile* transport is used for local delivery to user mailboxes in traditional BSD mailbox format. By default it runs under the uid and gid of the local user, which requires the sticky bit to be set on the */var/mail* directory. Some systems use the alternative approach of running mail deliveries under a particular group instead of using the sticky bit. The commented options show how this can be done.

Exim adds three headers to the message as it delivers it: *Delivery-date:*, *Envelope-to:* and *Return-path:*. This action is requested by the three similarly-named options above.

```
address_pipe:
  driver = pipe
  return_output
```

This transport is used for handling deliveries to pipes that are generated by redirection (aliasing or users' *.forward* files). The **return_output** option specifies that any output generated by the pipe is to be returned to the sender.

```
address_file:
  driver = appendfile
  delivery_date_add
  envelope_to_add
  return_path_add
```

This transport is used for handling deliveries to files that are generated by redirection. The name of the file is not specified in this instance of *appendfile*, because it comes from the *redirect* router.

```
address_reply:
  driver = autoreply
```

This transport is used for handling automatic replies generated by users' filter files.

7.5 Default retry rule

The retry section of the configuration file contains rules which affect the way Exim retries deliveries that cannot be completed at the first attempt. It is introduced by the line

```
begin retry
```

In the default configuration, there is just one rule, which applies to all errors:

```
*   *   F,2h,15m; G,16h,1h,1.5; F,4d,6h
```

This causes any temporarily failing address to be retried every 15 minutes for 2 hours, then at intervals starting at one hour and increasing by a factor of 1.5 until 16 hours have passed, then every 6 hours up to 4 days. If an address is not delivered after 4 days of temporary failure, it is bounced.

If the retry section is removed from the configuration, or is empty (that is, if no retry rules are defined), Exim will not retry deliveries. This turns temporary errors into permanent errors.

7.6 Rewriting configuration

The rewriting section of the configuration, introduced by

```
begin rewrite
```

contains rules for rewriting addresses in messages as they arrive. There are no rewriting rules in the default configuration file.

7.7 Authenticators configuration

The authenticators section of the configuration, introduced by

```
begin authenticators
```

defines mechanisms for the use of the SMTP AUTH command. The default configuration file contains two commented-out example authenticators which support plaintext username/password authentication using the standard PLAIN mechanism and the traditional but non-standard LOGIN mechanism, with Exim acting as the server. PLAIN and LOGIN are enough to support most MUA software.

The example PLAIN authenticator looks like this:

```
#PLAIN:
#  driver                  = plaintext
#  server_set_id           = $auth2
#  server_prompts          = :
#  server_condition        = Authentication is not yet configured
#  server_advertise_condition = ${if def:tls_cipher }
```

And the example LOGIN authenticator looks like this:

```
#LOGIN:
#  driver                  = plaintext
#  server_set_id           = $auth1
#  server_prompts          = <| Username: | Password:
#  server_condition        = Authentication is not yet configured
#  server_advertise_condition = ${if def:tls_cipher }
```

The **server_set_id** option makes Exim remember the authenticated username in *$authenticated_id*, which can be used later in ACLs or routers. The **server_prompts** option configures the *plaintext* authenticator so that it implements the details of the specific authentication mechanism, i.e. PLAIN or LOGIN. The **server_advertise_condition** setting controls when Exim offers authentication to clients; in the examples, this is only when TLS or SSL has been started, so to enable the authenticators you also need to add support for TLS as described in 7.1.

The **server_condition** setting defines how to verify that the username and password are correct. In the examples it just produces an error message. To make the authenticators work, you can use a string expansion expression like one of the examples in 34.

8. Regular expressions

Exim supports the use of regular expressions in many of its options. It uses the PCRE regular expression library; this provides regular expression matching that is compatible with Perl 5. The syntax and semantics of regular expressions is discussed in many Perl reference books, and also in Jeffrey Friedl's *Mastering Regular Expressions*, which is published by O'Reilly (see **http://www.oreilly.com/catalog/regex2/**).

The documentation for the syntax and semantics of the regular expressions that are supported by PCRE is included in plain text in the file *doc/pcrepattern.txt* in the Exim distribution, and also in the HTML tarbundle of Exim documentation. It describes in detail the features of the regular expressions that PCRE supports, so no further description is included here. The PCRE functions are called from Exim using the default option settings (that is, with no PCRE options set), except that the PCRE_ CASELESS option is set when the matching is required to be case-insensitive.

In most cases, when a regular expression is required in an Exim configuration, it has to start with a circumflex, in order to distinguish it from plain text or an "ends with" wildcard. In this example of a configuration setting, the second item in the colon-separated list is a regular expression.

```
domains = a.b.c : ^\\d{3} : *.y.z : ...
```

The doubling of the backslash is required because of string expansion that precedes interpretation – see section 11.1 for more discussion of this issue, and a way of avoiding the need for doubling backslashes. The regular expression that is eventually used in this example contains just one backslash. The circumflex is included in the regular expression, and has the normal effect of "anchoring" it to the start of the string that is being matched.

There are, however, two cases where a circumflex is not required for the recognition of a regular expression: these are the **match** condition in a string expansion, and the **matches** condition in an Exim filter file. In these cases, the relevant string is always treated as a regular expression; if it does not start with a circumflex, the expression is not anchored, and can match anywhere in the subject string.

In all cases, if you want a regular expression to match at the end of a string, you must code the $ metacharacter to indicate this. For example:

```
domains = ^\\d{3}\\.example
```

matches the domain *123.example*, but it also matches *123.example.com*. You need to use:

```
domains = ^\\d{3}\\.example\$
```

if you want *example* to be the top-level domain. The backslash before the $ is needed because string expansion also interprets dollar characters.

8.1 Testing regular expressions

A program called *pcretest* forms part of the PCRE distribution and is built with PCRE during the process of building Exim. It is primarily intended for testing PCRE itself, but it can also be used for experimenting with regular expressions. After building Exim, the binary can be found in the build directory (it is not installed anywhere automatically). There is documentation of various options in *doc/pcretest.txt*, but for simple testing, none are needed. This is the output of a sample run of *pcretest*:

```
  re> /^([@]+)@.+\.(ac|edu)\.(?!kr)[a-z]{2}$/
data> x@y.ac.uk
 0: x@y.ac.uk
 1: x
 2: ac
data> x@y.ac.kr
No match
data> x@y.edu.com
No match
data> x@y.edu.co
```

```
0: x@y.edu.co
1: x
2: edu
```

Input typed by the user is shown in bold face. After the “re>” prompt, a regular expression enclosed in delimiters is expected. If this compiles without error, “data>” prompts are given for strings against which the expression is matched. An empty data line causes a new regular expression to be read. If the match is successful, the captured substring values (that is, what would be in the variables *$0*, *$1*, *$2*, etc.) are shown. The above example tests for an email address whose domain ends with either “ac” or “edu” followed by a two-character top-level domain that is not “kr”. The local part is captured in *$1* and the “ac” or “edu” in *$2*.

9. File and database lookups

Exim can be configured to look up data in files or databases as it processes messages. Two different kinds of syntax are used:

(1) A string that is to be expanded may contain explicit lookup requests. These cause parts of the string to be replaced by data that is obtained from the lookup. Lookups of this type are conditional expansion items. Different results can be defined for the cases of lookup success and failure. See chapter 11, where string expansions are described in detail.

(2) Lists of domains, hosts, and email addresses can contain lookup requests as a way of avoiding excessively long linear lists. In this case, the data that is returned by the lookup is often (but not always) discarded; whether the lookup succeeds or fails is what really counts. These kinds of list are described in chapter 10.

String expansions, lists, and lookups interact with each other in such a way that there is no order in which to describe any one of them that does not involve references to the others. Each of these three chapters makes more sense if you have read the other two first. If you are reading this for the first time, be aware that some of it will make a lot more sense after you have read chapters 10 and 11.

9.1 Examples of different lookup syntax

It is easy to confuse the two different kinds of lookup, especially as the lists that may contain the second kind are always expanded before being processed as lists. Therefore, they may also contain lookups of the first kind. Be careful to distinguish between the following two examples:

```
domains = ${lookup{$sender_host_address}lsearch{/some/file}}
domains = lsearch;/some/file
```

The first uses a string expansion, the result of which must be a domain list. No strings have been specified for a successful or a failing lookup; the defaults in this case are the looked-up data and an empty string, respectively. The expansion takes place before the string is processed as a list, and the file that is searched could contain lines like this:

```
192.168.3.4: domain1:domain2:...
192.168.1.9: domain3:domain4:...
```

When the lookup succeeds, the result of the expansion is a list of domains (and possibly other types of item that are allowed in domain lists).

In the second example, the lookup is a single item in a domain list. It causes Exim to use a lookup to see if the domain that is being processed can be found in the file. The file could contains lines like this:

```
domain1:
domain2:
```

Any data that follows the keys is not relevant when checking that the domain matches the list item.

It is possible, though no doubt confusing, to use both kinds of lookup at once. Consider a file containing lines like this:

```
192.168.5.6: lsearch;/another/file
```

If the value of *$sender_host_address* is 192.168.5.6, expansion of the first **domains** setting above generates the second setting, which therefore causes a second lookup to occur.

The rest of this chapter describes the different lookup types that are available. Any of them can be used in any part of the configuration where a lookup is permitted.

9.2 Lookup types

Two different types of data lookup are implemented:

- The *single-key* type requires the specification of a file in which to look, and a single key to search for. The key must be a non-empty string for the lookup to succeed. The lookup type determines how the file is searched.
- The *query-style* type accepts a generalized database query. No particular key value is assumed by Exim for query-style lookups. You can use whichever Exim variables you need to construct the database query.

The code for each lookup type is in a separate source file that is included in the binary of Exim only if the corresponding compile-time option is set. The default settings in *src/EDITME* are:

```
LOOKUP_DBM=yes
LOOKUP_LSEARCH=yes
```

which means that only linear searching and DBM lookups are included by default. For some types of lookup (e.g. SQL databases), you need to install appropriate libraries and header files before building Exim.

9.3 Single-key lookup types

The following single-key lookup types are implemented:

- *cdb*: The given file is searched as a Constant DataBase file, using the key string without a terminating binary zero. The cdb format is designed for indexed files that are read frequently and never updated, except by total re-creation. As such, it is particularly suitable for large files containing aliases or other indexed data referenced by an MTA. Information about cdb can be found in several places:

  http://www.pobox.com/~djb/cdb.html
  ftp://ftp.corpit.ru/pub/tinycdb/
  http://packages.debian.org/stable/utils/freecdb.html

  A cdb distribution is not needed in order 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, so you need to obtain a cdb distribution in order to do this.

- *dbm*: Calls to DBM library functions are used to extract data from the given DBM file by looking up the record with the given key. A terminating binary zero is included in the key that is passed to the DBM library. See section 4.3 for a discussion of DBM libraries.

  For all versions of Berkeley DB, Exim uses the DB_HASH style of database when building DBM files using the **exim_dbmbuild** utility. However, when using Berkeley DB versions 3 or 4, it opens existing databases for reading with the DB_UNKNOWN option. This enables it to handle any of the types of database that the library supports, and can be useful for accessing DBM files created by other applications. (For earlier DB versions, DB_HASH is always used.)

- *dbmnz*: This is the same as *dbm*, except that a terminating binary zero is not included in the key that is passed to the DBM library. You may need this if you want to look up data in files that are created by or shared with some other application that does not use terminating zeros. For example, you need to use *dbmnz* rather than *dbm* if you want to authenticate incoming SMTP calls using the passwords from Courier's */etc/userdbshadow.dat* file. Exim's utility program for creating DBM files (*exim_dbmbuild*) includes the zeros by default, but has an option to omit them (see section 50.9).

- *dsearch*: The given file must be a directory; this is searched for an entry whose name is the key by calling the *lstat()* function. The key may not contain any forward slash characters. If *lstat()* succeeds, the result of the lookup is the name of the entry, which may be a file, directory, symbolic link, or any other kind of directory entry. An example of how this lookup can be used to support virtual domains is given in section 47.7.

- *iplsearch*: The given file is a text file containing keys and data. A key is terminated by a colon or white space or the end of the line. The keys in the file must be IP addresses, or IP addresses with CIDR masks. Keys that involve IPv6 addresses must be enclosed in quotes to prevent the first internal colon being interpreted as a key terminator. For example:

```
1.2.3.4:            data for 1.2.3.4
192.168.0.0/16      data for 192.168.0.0/16
"abcd::cdab":       data for abcd::cdab
"abcd:abcd::/32"    data for abcd:abcd::/32
```

The key for an *iplsearch* lookup must be an IP address (without a mask). The file is searched linearly, using the CIDR masks where present, until a matching key is found. The first key that matches is used; there is no attempt to find a "best" match. Apart from the way the keys are matched, the processing for *iplsearch* is the same as for *lsearch*.

Warning 1: Unlike most other single-key lookup types, a file of data for *iplsearch* can *not* be turned into a DBM or cdb file, because those lookup types support only literal keys.

Warning 2: In a host list, you must always use *net-iplsearch* so that the implicit key is the host's IP address rather than its name (see section 10.12).

- *lsearch*: The given file is a text file that is searched linearly for a line beginning with the search key, terminated by a colon or white space or the end of the line. The search is case-insensitive; that is, upper and lower case letters are treated as the same. The first occurrence of the key that is found in the file is used.

  White space between the key and the colon is permitted. The remainder of the line, with leading and trailing white space removed, is the data. This can be continued onto subsequent lines by starting them with any amount of white space, but only a single space character is included in the data at such a junction. If the data begins with a colon, the key must be terminated by a colon, for example:

  ```
  baduser:  :fail:
  ```

  Empty lines and lines beginning with # are ignored, even if they occur in the middle of an item. This is the traditional textual format of alias files. Note that the keys in an *lsearch* file are literal strings. There is no wildcarding of any kind.

  In most *lsearch* files, keys are not required to contain colons or # characters, or white space. However, if you need this feature, it is available. If a key begins with a doublequote character, it is terminated only by a matching quote (or end of line), and the normal escaping rules apply to its contents (see section 6.16). An optional colon is permitted after quoted keys (exactly as for unquoted keys). There is no special handling of quotes for the data part of an *lsearch* line.

- *nis*: The given file is the name of a NIS map, and a NIS lookup is done with the given key, without a terminating binary zero. There is a variant called *nis0* which does include the terminating binary zero in the key. This is reportedly needed for Sun-style alias files. Exim does not recognize NIS aliases; the full map names must be used.

- *wildlsearch* or *nwildlsearch*: These search a file linearly, like *lsearch*, but instead of being interpreted as a literal string, each key in the file may be wildcarded. The difference between these two lookup types is that for *wildlsearch*, each key in the file is string-expanded before being used, whereas for *nwildlsearch*, no expansion takes place.

  Like *lsearch*, the testing is done case-insensitively. However, keys in the file that are regular expressions can be made case-sensitive by the use of `(-i)` within the pattern. The following forms of wildcard are recognized:

  (1) The string may begin with an asterisk to mean "ends with". For example:

  ```
  *.a.b.c       data for anything.a.b.c
  *fish         data for anythingfish
  ```

  (2) The string may begin with a circumflex to indicate a regular expression. For example, for *wildlsearch*:

  ```
  ^\N\d+\.a\.b\N    data for <digits>.a.b
  ```

  Note the use of `\N` to disable expansion of the contents of the regular expression. If you are using *nwildlsearch*, where the keys are not string-expanded, the equivalent entry is:

```
^\d+\.a\.b        data for <digits>.a.b
```

The case-insensitive flag is set at the start of compiling the regular expression, but it can be turned off by using `(-i)` at an appropriate point. For example, to make the entire pattern case-sensitive:

```
^(?-i)\d+\.a\.b        data for <digits>.a.b
```

If the regular expression contains white space or colon characters, you must either quote it (see *lsearch* above), or represent these characters in other ways. For example, `\s` can be used for white space and `\x3A` for a colon. This may be easier than quoting, because if you quote, you have to escape all the backslashes inside the quotes.

Note: It is not possible to capture substrings in a regular expression match for later use, because the results of all lookups are cached. If a lookup is repeated, the result is taken from the cache, and no actual pattern matching takes place. The values of all the numeric variables are unset after a *(n)wildlsearch* match.

(3) Although I cannot see it being of much use, the general matching function that is used to implement *(n)wildlsearch* means that the string may begin with a lookup name terminated by a semicolon, and followed by lookup data. For example:

```
cdb;/some/file  data for keys that match the file
```

The data that is obtained from the nested lookup is discarded.

Keys that do not match any of these patterns are interpreted literally. The continuation rules for the data are the same as for *lsearch*, and keys may be followed by optional colons.

Warning: Unlike most other single-key lookup types, a file of data for *(n)wildlsearch* can *not* be turned into a DBM or cdb file, because those lookup types support only literal keys.

9.4 Query-style lookup types

The supported query-style lookup types are listed below. Further details about many of them are given in later sections.

- *dnsdb*: This does a DNS search for one or more records whose domain names are given in the supplied query. The resulting data is the contents of the records. See section 9.10.
- *ibase*: This does a lookup in an InterBase database.
- *ldap*: This does an LDAP lookup using a query in the form of a URL, and returns attributes from a single entry. There is a variant called *ldapm* that permits values from multiple entries to be returned. A third variant called *ldapdn* returns the Distinguished Name of a single entry instead of any attribute values. See section 9.13.
- *mysql*: The format of the query is an SQL statement that is passed to a MySQL database. See section 9.20.
- *nisplus*: This does a NIS+ lookup using a query that can specify the name of the field to be returned. See section 9.19.
- *oracle*: The format of the query is an SQL statement that is passed to an Oracle database. See section 9.20.
- *passwd* is a query-style lookup with queries that are just user names. The lookup calls *getpwnam()* to interrogate the system password data, and on success, the result string is the same as you would get from an *lsearch* lookup on a traditional */etc/passwd file*, though with `*` for the password value. For example:

```
*:42:42:King Rat:/home/kr:/bin/bash
```

- *pgsql*: The format of the query is an SQL statement that is passed to a PostgreSQL database. See section 9.20.

- *sqlite*: The format of the query is a file name followed by an SQL statement that is passed to an SQLite database. See section 9.25.
- *testdb*: This is a lookup type that is used for testing Exim. It is not likely to be useful in normal operation.
- *whoson*: *Whoson* (**http://whoson.sourceforge.net**) is a protocol that allows a server to check whether a particular (dynamically allocated) IP address is currently allocated to a known (trusted) user and, optionally, to obtain the identity of the said user. For SMTP servers, *Whoson* was popular at one time for "POP before SMTP" authentication, but that approach has been superseded by SMTP authentication. In Exim, *Whoson* can be used to implement "POP before SMTP" checking using ACL statements such as

  ```
  require condition = \
    ${lookup whoson {$sender_host_address}{yes}{no}}
  ```

  The query consists of a single IP address. The value returned is the name of the authenticated user, which is stored in the variable *$value*. However, in this example, the data in *$value* is not used; the result of the lookup is one of the fixed strings "yes" or "no".

9.5 Temporary errors in lookups

Lookup functions can return temporary error codes if the lookup cannot be completed. For example, an SQL or LDAP database might be unavailable. For this reason, it is not advisable to use a lookup that might do this for critical options such as a list of local domains.

When a lookup cannot be completed in a router or transport, delivery of the message (to the relevant address) is deferred, as for any other temporary error. In other circumstances Exim may assume the lookup has failed, or may give up altogether.

9.6 Default values in single-key lookups

In this context, a "default value" is a value specified by the administrator that is to be used if a lookup fails.

Note: This section applies only to single-key lookups. For query-style lookups, the facilities of the query language must be used. An attempt to specify a default for a query-style lookup provokes an error.

If "*" is added to a single-key lookup type (for example, **lsearch***) and the initial lookup fails, the key "*" is looked up in the file to provide a default value. See also the section on partial matching below.

Alternatively, if "*@" is added to a single-key lookup type (for example **dbm*@**) then, if the initial lookup fails and the key contains an @ character, a second lookup is done with everything before the last @ replaced by *. This makes it possible to provide per-domain defaults in alias files that include the domains in the keys. If the second lookup fails (or doesn't take place because there is no @ in the key), "*" is looked up. For example, a *redirect* router might contain:

```
data = ${lookup{$local_part@$domain}lsearch*@{/etc/mix-aliases}}
```

Suppose the address that is being processed is *jane@eyre.example*. Exim looks up these keys, in this order:

```
jane@eyre.example
*@eyre.example
*
```

The data is taken from whichever key it finds first. **Note**: In an *lsearch* file, this does not mean the first of these keys in the file. A complete scan is done for each key, and only if it is not found at all does Exim move on to try the next key.

9.7 Partial matching in single-key lookups

The normal operation of a single-key lookup is to search 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. In this case, information in the file that has a key starting with "*." is matched by any domain that ends with the components that follow the full stop. For example, if a key in a DBM file is

```
*.dates.fict.example
```

then when partial matching is enabled this is matched by (amongst others) *2001.dates.fict.example* and *1984.dates.fict.example*. It is also matched by *dates.fict.example*, if that does not appear as a separate key in the file.

Note: Partial matching is not available for query-style lookups. It is also not available for any lookup items in address lists (see section 10.19).

Partial matching is implemented by doing a series of separate lookups using keys constructed by modifying the original subject key. This means that it can be used with any of the single-key lookup types, provided that partial matching keys beginning with a special prefix (default "*.") are included in the data file. Keys in the file that do not begin with the prefix are matched only by unmodified subject keys when partial matching is in use.

Partial matching is requested by adding the string "partial-" to the front of the name of a single-key lookup type, for example, **partial-dbm**. When this is done, the subject key is first looked up unmodified; if that fails, "*." is added at the start of the subject key, and it is looked up again. If that fails, further lookups are tried with dot-separated components removed from the start of the subject key, one-by-one, and "*." added on the front of what remains.

A minimum number of two non-* components are required. This can be adjusted by including a number before the hyphen in the search type. For example, **partial3-lsearch** specifies a minimum of three non-* components in the modified keys. Omitting the number is equivalent to "partial2-". If the subject key is *2250.dates.fict.example* then the following keys are looked up when the minimum number of non-* components is two:

```
2250.dates.fict.example
*.2250.dates.fict.example
*.dates.fict.example
*.fict.example
```

As soon as one key in the sequence is successfully looked up, the lookup finishes.

The use of "*." as the partial matching prefix is a default that can be changed. The motivation for this feature is to allow Exim to operate with file formats that are used by other MTAs. A different prefix can be supplied in parentheses instead of the hyphen after "partial". For example:

```
domains = partial(.)lsearch;/some/file
```

In this example, if the domain is *a.b.c*, the sequence of lookups is `a.b.c`, `.a.b.c`, and `.b.c` (the default minimum of 2 non-wild components is unchanged). The prefix may consist of any punctuation characters other than a closing parenthesis. It may be empty, for example:

```
domains = partial1()cdb;/some/file
```

For this example, if the domain is *a.b.c*, the sequence of lookups is `a.b.c`, `b.c`, and `c`.

If "partial0" is specified, what happens at the end (when the lookup with just one non-wild component has failed, and the original key is shortened right down to the null string) depends on the prefix:

- If the prefix has zero length, the whole lookup fails.
- If the prefix has length 1, a lookup for just the prefix is done. For example, the final lookup for "partial0(.)" is for `.` alone.

- Otherwise, if the prefix ends in a dot, the dot is removed, and the remainder is looked up. With the default prefix, therefore, the final lookup is for "*" on its own.
- Otherwise, the whole prefix is looked up.

If the search type ends in "*" or "*@" (see section 9.6 above), 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 with the default prefix, because the "*" key is already included in the sequence of partial lookups. However, there might be a use for lookup types such as "partial0(.)lsearch*".

The use of "*" in lookup partial matching differs from its use as a wildcard in domain lists and the like. Partial matching works only in terms of dot-separated components; a key such as `*fict.example` in a database file is useless, because the asterisk in a partial matching subject key is always followed by a dot.

9.8 Lookup caching

Exim caches all lookup results in order to avoid needless repetition of lookups. However, because (apart from the daemon) Exim operates as a collection of independent, short-lived processes, this caching applies only within a single Exim process. There is no inter-process lookup caching facility.

For single-key lookups, Exim keeps the relevant files open in case there is another lookup that needs them. 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.

The single-key lookup files are closed and the lookup caches are flushed at strategic points during delivery – for example, after all routing is complete.

9.9 Quoting lookup data

When data from an incoming message is included in a query-style lookup, there is the possibility of special characters in the data messing up the syntax of 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. For NIS+, data can be enclosed in double quotes like this:

```
[name="$local_part"]
```

but this still 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 of the following form is provided:

```
${quote_<lookup-type>:<string>}
```

For example, the safest way to write the NIS+ query is

```
[name="${quote_nisplus:$local_part}"]
```

See chapter 11 for full coverage of string expansions. The quote operator can be used for all lookup types, but has no effect for single-key lookups, since no quoting is ever needed in their key strings.

9.10 More about dnsdb

The *dnsdb* lookup type uses the DNS as its database. A simple query consists of a record type and a domain name, separated by an equals sign. For example, an expansion string could contain:

```
${lookup dnsdb{mx=a.b.example}{$value}fail}
```

If the lookup succeeds, the result is placed in *$value*, which in this case is used on its own as the result. If the lookup does not succeed, the `fail` keyword causes a *forced expansion failure* – see section 11.4 for an explanation of what this means.

The supported DNS record types are A, CNAME, MX, NS, PTR, SRV, and TXT, and, when Exim is compiled with IPv6 support, AAAA (and A6 if that is also configured). If no type is given, TXT is assumed. When the type is PTR, the data can be an IP address, written as normal; inversion and the addition of **in-addr.arpa** or **ip6.arpa** happens automatically. For example:

```
${lookup dnsdb{ptr=192.168.4.5}{$value}fail}
```

If the data for a PTR record is not a syntactically valid IP address, it is not altered and nothing is added.

For an MX lookup, both the preference value and the host name are returned for each record, separated by a space. For an SRV lookup, the priority, weight, port, and host name are returned for each record, separated by spaces.

For any record type, if multiple records are found (or, for A6 lookups, if a single record leads to multiple addresses), the data is returned as a concatenation, with newline as the default separator. The order, of course, depends on the DNS resolver. You can specify a different separator character between multiple records by putting a right angle-bracket followed immediately by the new separator at the start of the query. For example:

```
${lookup dnsdb{>: a=host1.example}}
```

It is permitted to specify a space as the separator character. Further white space is ignored.

9.11 Pseudo dnsdb record types

By default, both the preference value and the host name are returned for each MX record, separated by a space. If you want only host names, you can use the pseudo-type MXH:

```
${lookup dnsdb{mxh=a.b.example}}
```

In this case, the preference values are omitted, and just the host names are returned.

Another pseudo-type is ZNS (for "zone NS"). It performs a lookup for NS records on the given domain, but if none are found, it removes the first component of the domain name, and tries again. This process continues until NS records are found or there are no more components left (or there is a DNS error). In other words, it may return the name servers for a top-level domain, but it never returns the root name servers. If there are no NS records for the top-level domain, the lookup fails. Consider these examples:

```
${lookup dnsdb{zns=xxx.quercite.com}}
${lookup dnsdb{zns=xxx.edu}}
```

Assuming that in each case there are no NS records for the full domain name, the first returns the name servers for **quercite.com**, and the second returns the name servers for **edu**.

You should be careful about how you use this lookup because, unless the top-level domain does not exist, the lookup always returns some host names. The sort of use to which this might be put is for seeing if the name servers for a given domain are on a blacklist. You can probably assume that the name servers for the high-level domains such as **com** or **co.uk** are not going to be on such a list.

A third pseudo-type is CSA (Client SMTP Authorization). This looks up SRV records according to the CSA rules, which are described in section 40.46. Although *dnsdb* supports SRV lookups directly, this is not sufficient because of the extra parent domain search behaviour of CSA. The result of a successful lookup such as:

```
${lookup dnsdb {csa=$sender_helo_name}}
```

has two space-separated fields: an authorization code and a target host name. The authorization code can be "Y" for yes, "N" for no, "X" for explicit authorization required but absent, or "?" for unknown.

9.12 Multiple dnsdb lookups

In the previous sections, *dnsdb* lookups for a single domain are described. However, you can specify a list of domains or IP addresses in a single *dnsdb* lookup. The list is specified in the normal Exim way, with colon as the default separator, but with the ability to change this. For example:

```
${lookup dnsdb{one.domain.com:two.domain.com}}
${lookup dnsdb{a=one.host.com:two.host.com}}
${lookup dnsdb{ptr = <; 1.2.3.4 ; 4.5.6.8}}
```

In order to retain backwards compatibility, there is one special case: if the lookup type is PTR and no change of separator is specified, Exim looks to see if the rest of the string is precisely one IPv6 address. In this case, it does not treat it as a list.

The data from each lookup is concatenated, with newline separators by default, in the same way that multiple DNS records for a single item are handled. A different separator can be specified, as described above.

The *dnsdb* lookup fails only if all the DNS lookups fail. If there is a temporary DNS error for any of them, the behaviour is controlled by an optional keyword followed by a comma that may appear before the record type. The possible keywords are "defer_strict", "defer_never", and "defer_lax". With "strict" behaviour, any temporary DNS error causes the whole lookup to defer. With "never" behaviour, a temporary DNS error is ignored, and the behaviour is as if the DNS lookup failed to find anything. With "lax" behaviour, all the queries are attempted, but a temporary DNS error causes the whole lookup to defer only if none of the other lookups succeed. The default is "lax", so the following lookups are equivalent:

```
${lookup dnsdb{defer_lax,a=one.host.com:two.host.com}}
${lookup dnsdb{a=one.host.com:two.host.com}}
```

Thus, in the default case, as long as at least one of the DNS lookups yields some data, the lookup succeeds.

9.13 More about LDAP

The original LDAP implementation came from the University of Michigan; this has become "Open LDAP", and there are now two different releases. Another implementation comes from Netscape, and Solaris 7 and subsequent releases contain inbuilt LDAP support. Unfortunately, though these are all compatible at the lookup function level, their error handling is different. For this reason it is necessary to set a compile-time variable when building Exim with LDAP, to indicate which LDAP library is in use. One of the following should appear in your *Local/Makefile*:

```
LDAP_LIB_TYPE=UMICHIGAN
LDAP_LIB_TYPE=OPENLDAP1
LDAP_LIB_TYPE=OPENLDAP2
LDAP_LIB_TYPE=NETSCAPE
LDAP_LIB_TYPE=SOLARIS
```

If LDAP_LIB_TYPE is not set, Exim assumes `OPENLDAP1`, which has the same interface as the University of Michigan version.

There are three LDAP lookup types in Exim. These behave slightly differently in the way they handle the results of a query:

- *ldap* requires the result to contain just one entry; if there are more, it gives an error.
- *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* permits the result to contain more than one entry; the attributes from all of them are returned.

For *ldap* and *ldapm*, if a query finds only entries with no attributes, Exim behaves as if the entry did not exist, and the lookup fails. The format of the data returned by a successful lookup is described in the next section. First we explain how LDAP queries are coded.

9.14 Format of LDAP queries

An LDAP query takes the form of a URL as defined in RFC 2255. For example, in the configuration of a *redirect* router one might have this setting:

```
data = ${lookup ldap \
  {ldap:///cn=$local_part,o=University%20of%20Cambridge,\
  c=UK?mailbox?base?}}
```

The URL may begin with `ldap` or `ldaps` if your LDAP library supports secure (encrypted) LDAP connections. The second of these ensures that an encrypted TLS connection is used.

9.15 LDAP quoting

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. Furthermore, within an LDAP query, two different kinds of quoting are required. For this reason, there are two different LDAP-specific quoting operators.

The **quote_ldap** operator is designed for use on strings that are part of filter specifications. Conceptually, it first does the following conversions on the string:

```
*    =>    \2A
(    =>    \28
)    =>    \29
\    =>    \5C
```

in accordance with RFC 2254. The resulting string is then quoted according to the rules for URLs, that is, all non-alphanumeric characters except

```
! $ ' - . _ ( ) * +
```

are converted to their hex values, preceded by a percent sign. For example:

```
${quote_ldap: a(bc)*, a<yz>; }
```

yields

```
%20a%5C28bc%5C29%5C2A%2C%20a%3Cyz%3E%3B%20
```

Removing the URL quoting, this is (with a leading and a trailing space):

```
a\28bc\29\2A, a<yz>;
```

The **quote_ldap_dn** operator is designed for use on strings that are part of base DN specifications in queries. Conceptually, it first converts the string by inserting a backslash in front of any of the following characters:

```
, + " \ < > ;
```

It also inserts a backslash before any leading spaces or # characters, and before any trailing spaces. (These rules are in RFC 2253.) The resulting string is then quoted according to the rules for URLs. For example:

```
${quote_ldap_dn: a(bc)*, a<yz>; }
```

yields

```
%5C%20a(bc)*%5C%2C%20a%5C%3Cyz%5C%3E%5C%3B%5C%20
```

Removing the URL quoting, this is (with a trailing space):

```
\ a(bc)*\, a\<yz\>\;\
```

There are some further comments about quoting in the section on LDAP authentication below.

9.16 LDAP connections

The connection to an LDAP server may either be over TCP/IP, or, when OpenLDAP is in use, via a Unix domain socket. The example given above does not specify an LDAP server. A server that is reached by TCP/IP can be specified in a query by starting it 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 which 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 multiple values when only a single value is expected. Errors which cause the next server to be tried are connection failures, bind failures, and timeouts.

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.

If you are using the OpenLDAP library, you can connect to an LDAP server using a Unix domain socket instead of a TCP/IP connection. This is specified by using `ldapi` instead of `ldap` in LDAP queries. What follows here applies only to OpenLDAP. If Exim is compiled with a different LDAP library, this feature is not available.

For this type of connection, instead of a host name for the server, a pathname for the socket is required, and the port number is not relevant. The pathname can be specified either as an item in **ldap_default_servers**, or inline in the query. In the former case, you can have settings such as

```
ldap_default_servers = /tmp/ldap.sock : backup.ldap.your.domain
```

When the pathname is given in the query, you have to escape the slashes as `%2F` to fit in with the LDAP URL syntax. For example:

```
${lookup ldap {ldapi://%2Ftmp%2Fldap.sock/o=...
```

When Exim processes an LDAP lookup and finds that the "hostname" is really a pathname, it uses the Unix domain socket code, even if the query actually specifies `ldap` or `ldaps`. In particular, no encryption is used for a socket connection. This behaviour means that you can use a setting of **ldap_default_servers** such as in the example above with traditional `ldap` or `ldaps` queries, and it will work. First, Exim tries a connection via the Unix domain socket; if that fails, it tries a TCP/IP connection to the backup host.

If an explicit `ldapi` type is given in a query when a host name is specified, an error is diagnosed. However, if there are more items in **ldap_default_servers**, they are tried. In other words:

- Using a pathname with `ldap` or `ldaps` forces the use of the Unix domain interface.
- Using `ldapi` with a host name causes an error.

Using `ldapi` with no host or path in the query, and no setting of **ldap_default_servers**, does whatever the library does by default.

9.17 LDAP authentication and control information

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, and when double quotes are used, backslash is interpreted in the usual way inside them. The following names are recognized:

DEREFERENCE	set the dereferencing parameter
NETTIME	set a timeout for a network operation
USER	set the DN, for authenticating the LDAP bind
PASS	set the password, likewise
REFERRALS	set the referrals parameter
SIZE	set the limit for the number of entries returned
TIME	set the maximum waiting time for a query

The value of the DEREFERENCE parameter must be one of the words "never", "searching", "finding", or "always". The value of the REFERRALS parameter must be "follow" (the default) or "nofollow". The latter stops the LDAP library from trying to follow referrals issued by the LDAP server.

The name CONNECT is an obsolete name for NETTIME, retained for backwards compatibility. This timeout (specified as a number of seconds) is enforced from the client end for operations that can be carried out over a network. Specifically, it applies to network connections and calls to the *ldap_result()* function. If the value is greater than zero, it is used if LDAP_OPT_NETWORK_TIMEOUT is defined in the LDAP headers (OpenLDAP), or if LDAP_X_OPT_CONNECT_TIMEOUT is defined in the LDAP headers (Netscape SDK 4.1). A value of zero forces an explicit setting of "no timeout" for Netscape SDK; for OpenLDAP no action is taken.

The TIME parameter (also a number of seconds) is passed to the server to set a server-side limit on the time taken to complete a search.

Here is an example of an LDAP query in an Exim lookup that uses some of these values. This is a single line, folded to fit on the page:

```
${lookup ldap
  {user="cn=manager,o=University of Cambridge,c=UK" pass=secret
  ldap:///o=University%20of%20Cambridge,c=UK?sn?sub?(cn=foo)}
  {$value}fail}
```

The encoding of spaces as `%20` is a URL thing which should not be done for any of the auxiliary data. Exim configuration settings that include lookups which contain password information should be preceded by "hide" to prevent non-admin users from using the **-bP** option to see their values.

The auxiliary data items may be given in any order. The default is no connection timeout (the system timeout is used), no user or password, no limit on the number of entries returned, and no time limit on queries.

When a DN is quoted in the USER= setting for LDAP authentication, Exim removes any URL quoting that it may contain before passing it LDAP. Apparently some libraries do this for themselves, but some do not. Removing the URL quoting has two advantages:

- It makes it possible to use the same **quote_ldap_dn** expansion for USER= DNs as with DNs inside actual queries.
- It permits spaces inside USER= DNs.

For example, a setting such as

```
USER=cn=${quote_ldap_dn:$1}
```

should work even if *$1* contains spaces.

Expanded data for the PASS= value should be quoted using the **quote** expansion operator, rather than the LDAP quote operators. The only reason this field needs quoting is to ensure that it conforms to the Exim syntax, which does not allow unquoted spaces. For example:

```
PASS=${quote:$3}
```

The LDAP authentication mechanism can be used to check passwords as part of SMTP authentication. See the **ldapauth** expansion string condition in chapter 11.

9.18 Format of data returned by LDAP

The *ldapdn* lookup type returns the Distinguished Name from a single entry as a sequence of values, for example

```
cn=manager, o=University of Cambridge, c=UK
```

The *ldap* lookup type generates an error if more than one entry matches the search filter, whereas *ldapm* permits this case, and inserts a newline in the result between the data from different entries. It is possible for multiple values to be returned for both *ldap* and *ldapm*, but in the former case you know that whatever values are returned all came from a single entry in the directory.

In the common case where you specify a single attribute in your LDAP query, the result is not quoted, and does not contain the attribute name. If the attribute has multiple values, they are separated by commas.

If you specify multiple attributes, the result contains space-separated, quoted strings, each preceded by the attribute name and an equals sign. Within the quotes, the quote character, backslash, and newline are escaped with backslashes, and commas are used to separate multiple values for the attribute. Apart from the escaping, the string within quotes takes the same form as the output when a single attribute is requested. Specifying no attributes is the same as specifying all of an entry's attributes.

Here are some examples of the output format. The first line of each pair is an LDAP query, and the second is the data that is returned. The attribute called **attr1** has two values, whereas **attr2** has only one value:

```
ldap:///o=base?attr1?sub?(uid=fred)
value1.1, value1.2

ldap:///o=base?attr2?sub?(uid=fred)
value two

ldap:///o=base?attr1,attr2?sub?(uid=fred)
attr1="value1.1, value1.2" attr2="value two"

ldap:///o=base??sub?(uid=fred)
objectClass="top" attr1="value1.1, value1.2" attr2="value two"
```

The **extract** operator in string expansions can be used to pick out individual fields from data that consists of *key*=*value* pairs. You can make use of Exim's **-be** option to run expansion tests and thereby check the results of LDAP lookups.

9.19 More about NIS+

NIS+ queries consist of a NIS+ *indexed name* followed by an optional colon and field name. If this is given, the result of a successful query is the contents of the named field; otherwise 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 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

```
Martin Guerre
```

with no quotes. A NIS+ lookup fails if NIS+ returns more than one table entry for the given indexed key. The effect of the **quote_nisplus** expansion operator is to double any quote characters within the text.

9.20 SQL lookups

Exim can support lookups in InterBase, MySQL, Oracle, PostgreSQL, and SQLite databases. Queries for these databases contain SQL statements, so an example might be

```
${lookup mysql{select mailbox from users where id='userx'}\
  {$value}fail}
```

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

```
${lookup pgsql{select home,name from users where id='userx'}\
  {$value}}
```

might be

```
home=/home/userx name="Mister X"
```

Empty values and values containing spaces are double quoted, with embedded quotes escaped by a backslash. If the result of the query contains just one field, the value is passed back verbatim, without a field name, for example:

```
Mister X
```

If the result of the query yields more than one row, it is all concatenated, with a newline between the data for each row.

9.21 More about MySQL, PostgreSQL, Oracle, and InterBase

If any MySQL, PostgreSQL, Oracle, or InterBase lookups are used, the **mysql_servers**, **pgsql_servers**, **oracle_servers**, or **ibase_servers** option (as appropriate) must be set to a colon-separated list of server information. (For MySQL and PostgreSQL only, the global option need not be set if all queries contain their own server information – see section 9.22.) Each item in the list is a slash-separated list of four items: host name, database name, user name, and password. In the case of Oracle, the host name field is used for the "service name", and the database name field is not used and should be empty. For example:

```
hide oracle_servers = oracle.plc.example//userx/abcdwxyz
```

Because password data is sensitive, you should always precede the setting with "hide", to prevent non-admin users from obtaining the setting via the **-bP** option. Here is an example where two MySQL servers are listed:

```
hide mysql_servers = localhost/users/root/secret:\
                     otherhost/users/root/othersecret
```

For MySQL and PostgreSQL, a host may be specified as <*name*>:<*port*> but because this is a colon-separated list, the colon has to be doubled. For each query, these parameter groups are tried in order until a connection is made and a query is successfully processed. The result of a query may be that no data is found, but that is still a successful query. In other words, the list of servers provides a backup facility, not a list of different places to look.

The **quote_mysql**, **quote_pgsql**, and **quote_oracle** expansion operators convert newline, tab, carriage return, and backspace to \n, \t, \r, and \b respectively, and the characters single-quote, double-quote, and backslash itself are escaped with backslashes. The **quote_pgsql** expansion operator, in addition, escapes the percent and underscore characters. This cannot be done for MySQL because these escapes are not recognized in contexts where these characters are not special.

9.22 Specifying the server in the query

For MySQL and PostgreSQL lookups (but not currently for Oracle and InterBase), it is possible to specify a list of servers with an individual query. This is done by starting the query with

```
servers=server1:server2:server3:...;
```

Each item in the list may take one of two forms:

(1) If it contains no slashes it is assumed to be just a host name. The appropriate global option (**mysql_servers** or **pgsql_servers**) is searched for a host of the same name, and the remaining parameters (database, user, password) are taken from there.

(2) If it contains any slashes, it is taken as a complete parameter set.

The list of servers is used in exactly the same way as the global list. Once a connection to a server has happened and a query has been successfully executed, processing of the lookup ceases.

This feature is intended for use in master/slave situations where updates are occurring and you want to update the master rather than a slave. If the master is in the list as a backup for reading, you might have a global setting like this:

```
mysql_servers = slave1/db/name/pw:\
                slave2/db/name/pw:\
                master/db/name/pw
```

In an updating lookup, you could then write:

```
${lookup mysql{servers=master; UPDATE ...}
```

That query would then be sent only to the master server. If, on the other hand, the master is not to be used for reading, and so is not present in the global option, you can still update it by a query of this form:

```
${lookup pgsql{servers=master/db/name/pw; UPDATE ...}
```

9.23 Special MySQL features

For MySQL, an empty host name or the use of "localhost" in **mysql_servers** causes a connection to the server on the local host by means of a Unix domain socket. An alternate socket can be specified in parentheses. The full syntax of each item in **mysql_servers** is:

<*hostname*>::<*port*>(<*socket name*>)/<*database*>/<*user*>/<*password*>

Any of the three sub-parts of the first field can be omitted. For normal use on the local host it can be left blank or set to just "localhost".

No database need be supplied – but if it is absent here, it must be given in the queries.

If a MySQL query is issued that does not request any data (an insert, update, or delete command), the result of the lookup is the number of rows affected.

Warning: This can be misleading. If an update does not actually change anything (for example, setting a field to the value it already has), the result is zero because no rows are affected.

9.24 Special PostgreSQL features

PostgreSQL lookups can also use Unix domain socket connections to the database. This is usually faster and costs less CPU time than a TCP/IP connection. However it can be used only if the mail server runs on the same machine as the database server. A configuration line for PostgreSQL via Unix domain sockets looks like this:

```
hide pgsql_servers = (/tmp/.s.PGSQL.5432)/db/user/password : ...
```

In other words, instead of supplying a host name, a path to the socket is given. The path name is enclosed in parentheses so that its slashes aren't visually confused with the delimiters for the other server parameters.

If a PostgreSQL query is issued that does not request any data (an insert, update, or delete command), the result of the lookup is the number of rows affected.

9.25 More about SQLite

SQLite is different to the other SQL lookups because a file name is required in addition to the SQL query. An SQLite database is a single file, and there is no daemon as in the other SQL databases. The interface to Exim requires the name of the file, as an absolute path, to be given at the start of the query. It is separated from the query by white space. This means that the path name cannot contain white space. Here is a lookup expansion example:

```
${lookup sqlite {/some/thing/sqlitedb \
  select name from aliases where id='userx';}}
```

In a list, the syntax is similar. For example:

```
domainlist relay_domains = sqlite;/some/thing/sqlitedb \
   select * from relays where ip='$sender_host_address';
```

The only character affected by the **quote_sqlite** operator is a single quote, which it doubles.

The SQLite library handles multiple simultaneous accesses to the database internally. Multiple readers are permitted, but only one process can update at once. Attempts to access the database while it is being updated are rejected after a timeout period, during which the SQLite library waits for the lock to be released. In Exim, the default timeout is set to 5 seconds, but it can be changed by means of the **sqlite_lock_timeout** option.

10. Domain, host, address, and local part lists

A number of Exim configuration options contain lists of domains, hosts, email addresses, or local parts. For example, the **hold_domains** option contains a list of domains whose delivery is currently suspended. These lists are also used as data in ACL statements (see chapter 40), and as arguments to expansion conditions such as **match_domain**.

Each item in one of these lists is a pattern to be matched against a domain, host, email address, or local part, respectively. In the sections below, the different types of pattern for each case are described, but first we cover some general facilities that apply to all four kinds of list.

10.1 Expansion of lists

Each list is expanded as a single string before it is used. The result of expansion must be a list, possibly containing empty items, which is split up into separate items for matching. By default, colon is the separator character, but this can be varied if necessary. See sections 6.19 and 6.21 for details of the list syntax; the second of these discusses the way to specify empty list items.

If the string expansion is forced to fail, Exim behaves as if the item it is testing (domain, host, address, or local part) is not in the list. Other expansion failures cause temporary errors.

If an item in a list is a regular expression, backslashes, dollars and possibly other special characters in the expression must be protected against misinterpretation by the string expander. The easiest way to do this is to use the `\N` expansion feature to indicate that the contents of the regular expression should not be expanded. For example, in an ACL you might have:

```
deny senders = \N^\d{8}\w@.*\.baddomain\.example$\N : \
               ${lookup{$domain}lsearch{/badsenders/bydomain}}
```

The first item is a regular expression that is protected from expansion by `\N`, whereas the second uses the expansion to obtain a list of unwanted senders based on the receiving domain.

10.2 Negated items in lists

Items in a list may be positive or negative. Negative items are indicated by a leading exclamation mark, which may be followed by optional white space. A list defines a set of items (domains, etc). When Exim processes one of these lists, it is trying to find out whether a domain, host, address, or local part (respectively) is in the set that is defined by the list. It works like this:

The list is scanned from left to right. If a positive item is matched, the subject that is being checked is in the set; if a negative item is matched, the subject is not in the set. If the end of the list is reached without the subject having matched any of the patterns, it is in the set if the last item was a negative one, but not if it was a positive one. For example, the list in

```
domainlist relay_domains = !a.b.c : *.b.c
```

matches any domain ending in *.b.c* except for *a.b.c*. Domains that match neither *a.b.c* nor **.b.c* do not match, because the last item in the list is positive. However, if the setting were

```
domainlist relay_domains = !a.b.c
```

then all domains other than *a.b.c* would match because the last item in the list is negative. In other words, a list that ends with a negative item behaves as if it had an extra item `:*` on the end.

Another way of thinking about positive and negative items in lists is to read the connector as “or” after a positive item and as “and” after a negative item.

10.3 File names in lists

If an item in a domain, host, address, or local part list is an absolute file name (beginning with a slash character), each line of the file is read and processed as if it were an independent item in the list, except that further file names are not allowed, and no expansion of the data from the file takes place. Empty lines in the file are ignored, and the file may also contain comment lines:

- For domain and host lists, if a # character appears anywhere in a line of the file, it and all following characters are ignored.
- Because local parts may legitimately contain # characters, a comment in an address list or local part list file is recognized only if # is preceded by white space or the start of the line. For example:

```
not#comment@x.y.z   # but this is a comment
```

Putting a file name in a list has the same effect as inserting each line of the file as an item in the list (blank lines and comments excepted). However, there is one important difference: the file is read each time the list is processed, so if its contents vary over time, Exim's behaviour changes.

If a file name is preceded by an exclamation mark, the sense of any match within the file is inverted. For example, if

```
hold_domains = !/etc/nohold-domains
```

and the file contains the lines

```
!a.b.c
*.b.c
```

then *a.b.c* is in the set of domains defined by **hold_domains**, whereas any domain matching `*.b.c` is not.

10.4 An lsearch file is not an out-of-line list

As will be described in the sections that follow, lookups can be used in lists to provide indexed methods of checking list membership. There has been some confusion about the way *lsearch* lookups work in lists. Because an *lsearch* file contains plain text and is scanned sequentially, it is sometimes thought that it is allowed to contain wild cards and other kinds of non-constant pattern. This is not the case. The keys in an *lsearch* file are always fixed strings, just as for any other single-key lookup type.

If you want to use a file to contain wild-card patterns that form part of a list, just give the file name on its own, without a search type, as described in the previous section. You could also use the *wildlsearch* or *nwildlsearch*, but there is no advantage in doing this.

10.5 Named lists

A list of domains, hosts, email addresses, or local parts can be given a name which is then used to refer to the list elsewhere in the configuration. This is particularly convenient if the same list is required in several different places. It also allows lists to be given meaningful names, which can improve the readability of the configuration. For example, it is conventional to define a domain list called *local_domains* for all the domains that are handled locally on a host, using a configuration line such as

```
domainlist local_domains = localhost:my.dom.example
```

Named lists are referenced by giving their name preceded by a plus sign, so, for example, a router that is intended to handle local domains would be configured with the line

```
domains = +local_domains
```

The first router in a configuration is often one that handles all domains except the local ones, using a configuration with a negated item like this:

```
dnslookup:
  driver = dnslookup
  domains = ! +local_domains
  transport = remote_smtp
  no_more
```

The four kinds of named list are created by configuration lines starting with the words **domainlist**, **hostlist**, **addresslist**, or **localpartlist**, respectively. Then there follows the name that you are defining, followed by an equals sign and the list itself. For example:

```
hostlist    relay_hosts = 192.168.23.0/24 : my.friend.example
addresslist bad_senders = cdb;/etc/badsenders
```

A named list may refer to other named lists:

```
domainlist  dom1 = first.example : second.example
domainlist  dom2 = +dom1 : third.example
domainlist  dom3 = fourth.example : +dom2 : fifth.example
```

Warning: If the last item in a referenced list is a negative one, the effect may not be what you intended, because the negation does not propagate out to the higher level. For example, consider:

```
domainlist  dom1 = !a.b
domainlist  dom2 = +dom1 : *.b
```

The second list specifies "either in the **dom1** list or **.b*". The first list specifies just "not *a.b*", so the domain *x.y* matches it. That means it matches the second list as well. The effect is not the same as

```
domainlist  dom2 = !a.b : *.b
```

where *x.y* does not match. It's best to avoid negation altogether in referenced lists if you can.

Named lists may have a performance advantage. When Exim is routing an address or checking an incoming message, it caches the result of tests on named lists. So, if you have a setting such as

```
domains = +local_domains
```

on several of your routers or in several ACL statements, the actual test is done only for the first one. However, the caching works only if there are no expansions within the list itself or any sublists that it references. In other words, caching happens only for lists that are known to be the same each time they are referenced.

By default, there may be up to 16 named lists of each type. This limit can be extended by changing a compile-time variable. The use of domain and host lists is recommended for concepts such as local domains, relay domains, and relay hosts. The default configuration is set up like this.

10.6 Named lists compared with macros

At first sight, named lists might seem to be no different from macros in the configuration file. However, macros are just textual substitutions. If you write

```
ALIST = host1 : host2
auth_advertise_hosts = !ALIST
```

it probably won't do what you want, because that is exactly the same as

```
auth_advertise_hosts = !host1 : host2
```

Notice that the second host name is not negated. However, if you use a host list, and write

```
hostlist alist = host1 : host2
auth_advertise_hosts = ! +alist
```

the negation applies to the whole list, and so that is equivalent to

```
auth_advertise_hosts = !host1 : !host2
```

10.7 Named list caching

While processing a message, Exim caches the result of checking a named list if it is sure that the list is the same each time. In practice, this means that the cache operates only if the list contains no $ characters, which guarantees that it will not change when it is expanded. Sometimes, however, you may have an expanded list that you know will be the same each time within a given message. For example:

```
domainlist special_domains = \
           ${lookup{$sender_host_address}cdb{/some/file}}
```

This provides a list of domains that depends only on the sending host's IP address. If this domain list is referenced a number of times (for example, in several ACL lines, or in several routers) the result of the check is not cached by default, because Exim does not know that it is going to be the same list each time.

By appending `_cache` to `domainlist` you can tell Exim to go ahead and cache the result anyway. For example:

```
domainlist_cache special_domains = ${lookup{...
```

If you do this, you should be absolutely sure that caching is going to do the right thing in all cases. When in doubt, leave it out.

10.8 Domain lists

Domain lists contain patterns that are to be matched against a mail domain. The following types of item may appear in domain lists:

- If a pattern consists of a single @ character, it matches the local host name, as set by the **primary_hostname** option (or defaulted). This makes it possible to use the same configuration file on several different hosts that differ only in their names.
- If a pattern consists of the string `@[]` it matches an IP address enclosed in square brackets (as in an email address that contains a domain literal), but only if that IP address is recognized as local for email routing purposes. The **local_interfaces** and **extra_local_interfaces** options can be used to control which of a host's several IP addresses are treated as local. In today's Internet, the use of domain literals is controversial.
- If a pattern consists of the string `@mx_any` it matches any domain that has an MX record pointing to the local host or to any host that is listed in **hosts_treat_as_local**. The items `@mx_primary` and `@mx_secondary` are similar, except that the first matches only when a primary MX target is the local host, and the second only when no primary MX target is the local host, but a secondary MX target is. "Primary" means an MX record with the lowest preference value – there may of course be more than one of them.

  The MX lookup that takes place when matching a pattern of this type is performed with the resolver options for widening names turned off. Thus, for example, a single-component domain will *not* be expanded by adding the resolver's default domain. See the **qualify_single** and **search_parents** options of the *dnslookup* router for a discussion of domain widening.

  Sometimes you may want to ignore certain IP addresses when using one of these patterns. You can specify this by following the pattern with `/ignore=`<*ip list*>, where <*ip list*> is a list of IP addresses. These addresses are ignored when processing the pattern (compare the **ignore_target_hosts** option on a router). For example:

  ```
  domains = @mx_any/ignore=127.0.0.1
  ```

  This example matches any domain that has an MX record pointing to one of the local host's IP addresses other than 127.0.0.1.

  The list of IP addresses is in fact processed by the same code that processes host lists, so it may contain CIDR-coded network specifications and it may also contain negative items.

  Because the list of IP addresses is a sublist within a domain list, you have to be careful about delimiters if there is more than one address. Like any other list, the default delimiter can be changed. Thus, you might have:

  ```
  domains = @mx_any/ignore=<;127.0.0.1;0.0.0.0 : \
            an.other.domain : ...
  ```

  so that the sublist uses semicolons for delimiters. When IPv6 addresses are involved, it is easiest to change the delimiter for the main list as well:

  ```
  domains = <? @mx_any/ignore=<;127.0.0.1;::1 ? \
            an.other.domain ? ...
  ```

- If a pattern starts with an asterisk, the remaining characters of the pattern are compared with the terminating characters of the domain. The use of "*" 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 item such as `*key.ex` matches *donkey.ex* as well as *cipher.key.ex*.

- If a pattern starts with a circumflex character, it is treated as a regular expression, and matched against the domain using a regular expression matching function. The circumflex is treated as part of the regular expression. Email domains are case-independent, so this regular expression match is by default case-independent, but you can make it case-dependent by starting it with `(?-i)`. References to descriptions of the syntax of regular expressions are given in chapter 8.

  Warning: Because domain lists are expanded before being processed, you must escape any backslash and dollar characters in the regular expression, or use the special `\N` sequence (see chapter 11) to specify that it is not to be expanded (unless you really do want to build a regular expression by expansion, of course).

- If a pattern starts with the name of a single-key lookup type followed by a semicolon (for example, "dbm;" or "lsearch;"), the remainder of the pattern must be a file name in a suitable format for the lookup type. For example, for "cdb;" it must be an absolute path:

  ```
  domains = cdb;/etc/mail/local_domains.cdb
  ```

  The appropriate type of lookup is done on the file using the domain name as the key. In most cases, the data that is looked up is not used; Exim is interested only in whether or not the key is present in the file. However, when a lookup is used for the **domains** option on a router or a **domains** condition in an ACL statement, the data is preserved in the *$domain_data* variable and can be referred to in other router options or other statements in the same ACL.

- Any of the single-key lookup type names may be preceded by `partial<`*n*`>-`, where the `<`*n*`>` is optional, for example,

  ```
  domains = partial-dbm;/partial/domains
  ```

  This causes partial matching logic to be invoked; a description of how this works is given in section 9.7.

- 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.

- If the pattern starts with the name of a query-style lookup type followed by a semicolon (for example, "nisplus;" or "ldap;"), the remainder of the pattern must be an appropriate query for the lookup type, as described in chapter 9. For example:

  ```
  hold_domains = mysql;select domain from holdlist \
    where domain = '$domain';
  ```

  In most cases, the data that is looked up is not used (so for an SQL query, for example, it doesn't matter what field you select). Exim is interested only in whether or not the query succeeds. However, when a lookup is used for the **domains** option on a router, the data is preserved in the *$domain_data* variable and can be referred to in other options.

- If none of the above cases apply, a caseless textual comparison is made between the pattern and the domain.

Here is an example that uses several different kinds of pattern:

```
domainlist funny_domains = \
  @ : \
  lib.unseen.edu : \
  *.foundation.fict.example : \
  \N^[1-2]\d{3}\.fict\.example$\N : \
```

```
partial-dbm;/opt/data/penguin/book : \
nis;domains.byname : \
nisplus;[name=$domain,status=local],domains.org_dir
```

There are obvious processing trade-offs among the various matching modes. Using an asterisk is faster than a regular expression, and listing a few names explicitly probably is too. The use of a file or database lookup is expensive, but may be the only option if hundreds of names are required. Because the patterns are tested in order, it makes sense to put the most commonly matched patterns earlier.

10.9 Host lists

Host lists are used to control what remote hosts are allowed to do. For example, some hosts may be allowed to use the local host as a relay, and some may be permitted to use the SMTP ETRN command. Hosts can be identified in two different ways, by name or by IP address. In a host list, some types of pattern are matched to a host name, and some are matched to an IP address. You need to be particularly careful with this when single-key lookups are involved, to ensure that the right value is being used as the key.

10.10 Special host list patterns

If a host list item is the empty string, it matches only when no remote host is involved. This is the case when a message is being received from a local process using SMTP on the standard input, that is, when a TCP/IP connection is not used.

The special pattern "*" in a host list matches any host or no host. Neither the IP address nor the name is actually inspected.

10.11 Host list patterns that match by IP address

If an IPv4 host calls an IPv6 host and the call is accepted on an IPv6 socket, 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. (Not all operating systems accept IPv4 calls on IPv6 sockets, as there have been some security concerns.)

The following types of pattern in a host list check the remote host by inspecting its IP address:

- If the pattern is a plain domain name (not a regular expression, not starting with *, not a lookup of any kind), Exim calls the operating system function to find the associated IP address(es). Exim uses the newer *getipnodebyname()* function when available, otherwise *gethostbyname()*. This typically causes a forward DNS lookup of the name. The result is compared with the IP address of the subject host.

  If there is a temporary problem (such as a DNS timeout) with the host name lookup, a temporary error occurs. For example, if the list is being used in an ACL condition, the ACL gives a "defer" response, usually leading to a temporary SMTP error code. If no IP address can be found for the host name, what happens is described in section 10.14 below.

- If the pattern is "@", the primary host name is substituted and used as a domain name, as just described.

- If the pattern is an IP address, it is matched against the IP address of the subject host. IPv4 addresses are given in the normal "dotted-quad" notation. IPv6 addresses can be given in colon-separated format, but the colons have to be doubled so as not to be taken as item separators when the default list separator is used. IPv6 addresses are recognized even when Exim is compiled without IPv6 support. This means that if they appear in a host list on an IPv4-only host, Exim will not treat them as host names. They are just addresses that can never match a client host.

- If the pattern is "@[]", it matches the IP address of any IP interface on the local host. For example, if the local host is an IPv4 host with one interface address 10.45.23.56, these two ACL statements have the same effect:

```
accept hosts = 127.0.0.1 : 10.45.23.56
accept hosts = @[]
```

- If the pattern is an IP address followed by a slash and a mask length (for example 10.11.42.0/24), it is matched against the IP address of the subject host under the given mask. This allows, an entire network of hosts to be included (or excluded) by a single item. The mask uses CIDR notation; it specifies the number of address bits that must match, starting from the most significant end of the address.

  Note: The mask is *not* a count of addresses, nor is it the high number of a range of addresses. It is the number of bits in the network portion of the address. The above example specifies a 24-bit netmask, so it matches all 256 addresses in the 10.11.42.0 network. An item such as

  ```
  192.168.23.236/31
  ```

  matches just two addresses, 192.168.23.236 and 192.168.23.237. A mask value of 32 for an IPv4 address is the same as no mask at all; just a single address matches.

  Here is another example which shows an IPv4 and an IPv6 network:

  ```
  recipient_unqualified_hosts = 192.168.0.0/16: \
                                3ffe::ffff::836f::::/48
  ```

  The doubling of list separator characters applies only when these items appear inline in a host list. It is not required when indirecting via a file. For example:

  ```
  recipient_unqualified_hosts = /opt/exim/unqualnets
  ```

  could make use of a file containing

  ```
  172.16.0.0/12
  3ffe:ffff:836f::/48
  ```

  to have exactly the same effect as the previous example. When listing IPv6 addresses inline, it is usually more convenient to use the facility for changing separator characters. This list contains the same two networks:

  ```
  recipient_unqualified_hosts = <; 172.16.0.0/12; \
                                   3ffe:ffff:836f::/48
  ```

  The separator is changed to semicolon by the leading "<;" at the start of the list.

10.12 Host list patterns for single-key lookups by host address

When a host is to be identified by a single-key lookup of its complete IP address, the pattern takes this form:

net-<*single-key-search-type*>;<*search-data*>

For example:

```
hosts_lookup = net-cdb;/hosts-by-ip.db
```

The text form of the IP address of the subject host is used as the lookup key. IPv6 addresses are converted to an unabbreviated form, using lower case letters, with dots as separators because colon is the key terminator in *lsearch* files. [Colons can in fact be used in keys in *lsearch* files by quoting the keys, but this is a facility that was added later.] The data returned by the lookup is not used.

Single-key lookups can also be performed using masked IP addresses, using patterns of this form:

net<*number*>-<*single-key-search-type*>;<*search-data*>

For example:

```
net24-dbm;/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 lookup key. For

example, if the host's IP address is 192.168.34.6, the key that is looked up for the above example is "192.168.34.0/24".

When an IPv6 address is converted to a string, dots are normally used instead of colons, so that keys in *lsearch* files need not contain colons (which terminate *lsearch* keys). This was implemented some time before the ability to quote keys was made available in *lsearch* files. However, the more recently implemented *iplsearch* files do require colons in IPv6 keys (notated using the quoting facility) so as to distinguish them from IPv4 keys. For this reason, when the lookup type is *iplsearch*, IPv6 addresses are converted using colons and not dots. In all cases, full, unabbreviated IPv6 addresses are always used.

Ideally, it would be nice to tidy up this anomalous situation by changing to colons in all cases, given that quoting is now available for *lsearch*. However, this would be an incompatible change that might break some existing configurations.

Warning: Specifying **net32-** (for an IPv4 address) or **net128-** (for an IPv6 address) is not the same as specifying just **net-** without a number. In the former case the key strings include the mask value, whereas in the latter case the IP address is used on its own.

10.13 Host list patterns that match by host name

There are several types of pattern that require Exim to know the name of the remote host. These are either wildcard patterns or lookups by name. (If a complete hostname is given without any wildcarding, it is used to find an IP address to match against, as described in the section 10.11 above.)

If the remote host name is not already known when Exim encounters one of these patterns, it has to be found from the IP address. Although many sites on the Internet are conscientious about maintaining reverse DNS data for their hosts, there are also many that do not do this. Consequently, a name cannot always be found, and this may lead to unwanted effects. Take care when configuring host lists with wildcarded name patterns. Consider what will happen if a name cannot be found.

Because of the problems of determining host names from IP addresses, matching against host names is not as common as matching against IP addresses.

By default, in order to find a host name, Exim first does a reverse DNS lookup; if no name is found in the DNS, the system function (*gethostbyaddr()* or *getipnodebyaddr()* if available) is tried. The order in which these lookups are done can be changed by setting the **host_lookup_order** option. For security, once Exim has found one or more names, it looks up the IP addresses for these names and compares them with the IP address that it started with. Only those names whose IP addresses match are accepted. Any other names are discarded. If no names are left, Exim behaves as if the host name cannot be found. In the most common case there is only one name and one IP address.

There are some options that control what happens if a host name cannot be found. These are described in section 10.14 below.

As a result of aliasing, hosts may have more than one name. When processing any of the following types of pattern, all the host's names are checked:

- If a pattern starts with "*" the remainder of the item must match the end of the host name. For example, `*.b.c` matches all hosts whose names end in *.b.c*. This special simple form is provided because this is a very common requirement. Other kinds of wildcarding require the use of a regular expression.
- If the item starts with "^" it is taken to be a regular expression which is matched against the host name. Host names are case-independent, so this regular expression match is by default case-independent, but you can make it case-dependent by starting it with `(?-i)`. References to descriptions of the syntax of regular expressions are given in chapter 8. For example,

  ```
  ^(a|b)\.c\.d$
  ```

  is a regular expression that matches either of the two hosts *a.c.d* or *b.c.d*. When a regular expression is used in a host list, you must take care that backslash and dollar characters are not misinterpreted as part of the string expansion. The simplest way to do this is to use `\N` to mark that part of the string as non-expandable. For example:

```
sender_unqualified_hosts = \N^(a|b)\.c\.d$\N : ....
```

Warning: If you want to match a complete host name, you must include the `$` terminating metacharacter in the regular expression, as in the above example. Without it, a match at the start of the host name is all that is required.

10.14 Behaviour when an IP address or name cannot be found

While processing a host list, Exim may need to look up an IP address from a name (see section 10.11), or it may need to look up a host name from an IP address (see section 10.13). In either case, the behaviour when it fails to find the information it is seeking is the same.

Note: This section applies to permanent lookup failures. It does *not* apply to temporary DNS errors, whose handling is described in the next section.

By default, Exim behaves as if the host does not match the list. This may not always be what you want to happen. To change Exim's behaviour, the special items `+include_unknown` or `+ignore_unknown` may appear in the list (at top level – they are not recognized in an indirected file).

- If any item that follows `+include_unknown` requires information that cannot found, Exim behaves as if the host does match the list. For example,

  ```
  host_reject_connection = +include_unknown:*.enemy.ex
  ```

  rejects connections from any host whose name matches `*.enemy.ex`, and also any hosts whose name it cannot find.

- If any item that follows `+ignore_unknown` requires information that cannot be found, Exim ignores that item and proceeds to the rest of the list. For example:

  ```
  accept hosts = +ignore_unknown : friend.example : \
                 192.168.4.5
  ```

  accepts from any host whose name is *friend.example* and from 192.168.4.5, whether or not its host name can be found. Without `+ignore_unknown`, if no name can be found for 192.168.4.5, it is rejected.

Both `+include_unknown` and `+ignore_unknown` may appear in the same list. The effect of each one lasts until the next, or until the end of the list.

10.15 Temporary DNS errors when looking up host information

A temporary DNS lookup failure normally causes a defer action (except when **dns_again_means_nonexist** converts it into a permanent error). However, host lists can include `+ignore_defer` and `+include_defer`, analagous to `+ignore_unknown` and `+include_unknown`, as described in the previous section. These options should be used with care, probably only in non-critical host lists such as whitelists.

10.16 Host list patterns for single-key lookups by host name

If a pattern is of the form

<*single-key-search-type*>;<*search-data*>

for example

```
dbm;/host/accept/list
```

a single-key lookup is performed, using the host name as its key. If the lookup succeeds, the host matches the item. The actual data that is looked up is not used.

Reminder: With this kind of pattern, you must have host *names* as keys in the file, not IP addresses. If you want to do lookups based on IP addresses, you must precede the search type with "net-" (see

section 10.12). There is, however, no reason why you could not use two items in the same list, one doing an address lookup and one doing a name lookup, both using the same file.

10.17 Host list patterns for query-style lookups

If a pattern is of the form

<*query-style-search-type*>;<*query*>

the query is obeyed, and if it succeeds, the host matches the item. The actual data that is looked up is not used. The variables *$sender_host_address* and *$sender_host_name* can be used in the query. For example:

```
hosts_lookup = pgsql;\
  select ip from hostlist where ip='$sender_host_address'
```

The value of *$sender_host_address* for an IPv6 address contains colons. You can use the **sg** expansion item to change this if you need to. If you want to use masked IP addresses in database queries, you can use the **mask** expansion operator.

If the query contains a reference to *$sender_host_name*, Exim automatically looks up the host name if has not already done so. (See section 10.13 for comments on finding host names.)

Historical note: prior to release 4.30, Exim would always attempt to find a host name before running the query, unless the search type was preceded by `net-`. This is no longer the case. For backwards compatibility, `net-` is still recognized for query-style lookups, but its presence or absence has no effect. (Of course, for single-key lookups, `net-` *is* important. See section 10.12.)

10.18 Mixing wildcarded host names and addresses in host lists

If you have name lookups or wildcarded host names and IP addresses in the same host list, you should normally put the IP addresses first. For example, in an ACL you could have:

```
accept hosts = 10.9.8.7 : *.friend.example
```

The reason for this lies in the left-to-right way that Exim processes lists. It can test IP addresses without doing any DNS lookups, but when it reaches an item that requires a host name, it fails if it cannot find a host name to compare with the pattern. If the above list is given in the opposite order, the **accept** statement fails for a host whose name cannot be found, even if its IP address is 10.9.8.7.

If you really do want to do the name check first, and still recognize the IP address, you can rewrite the ACL like this:

```
accept hosts = *.friend.example
accept hosts = 10.9.8.7
```

If the first **accept** fails, Exim goes on to try the second one. See chapter 40 for details of ACLs.

10.19 Address lists

Address lists contain patterns that are matched against mail addresses. There is one special case to be considered: the sender address of a bounce message is always empty. You can test for this by providing an empty item in an address list. For example, you can set up a router to process bounce messages by using this option setting:

```
senders = :
```

The presence of the colon creates an empty item. If you do not provide any data, the list is empty and matches nothing. The empty sender can also be detected by a regular expression that matches an empty string, and by a query-style lookup that succeeds when *$sender_address* is empty.

Non-empty items in an address list can be straightforward email addresses. For example:

```
senders = jbc@askone.example : hs@anacreon.example
```

A certain amount of wildcarding is permitted. If a pattern contains an @ character, but is not a regular expression and does not begin with a semicolon-terminated lookup type (described below), the local part of the subject address is compared with the local part of the pattern, which may start with an asterisk. If the local parts match, the domain is checked in exactly the same way as for a pattern in a domain list. For example, the domain can be wildcarded, refer to a named list, or be a lookup:

```
deny senders = *@*.spamming.site:\
               *@+hostile_domains:\
               bozo@partial-lsearch;/list/of/dodgy/sites:\
               *@dbm;/bad/domains.db
```

If a local part that begins with an exclamation mark is required, it has to be specified using a regular expression, because otherwise the exclamation mark is treated as a sign of negation, as is standard in lists.

If a non-empty pattern that is not a regular expression or a lookup does not contain an @ character, it is matched against the domain part of the subject address. The only two formats that are recognized this way are a literal domain, or a domain pattern that starts with *. In both these cases, the effect is the same as if `*@` preceded the pattern. For example:

```
deny senders = enemy.domain : *.enemy.domain
```

The following kinds of more complicated address list pattern can match any address, including the empty address that is characteristic of bounce message senders:

- If (after expansion) a pattern starts with "^", a regular expression match is done against the complete address, with the pattern as the regular expression. You must take care that backslash and dollar characters are not misinterpreted as part of the string expansion. The simplest way to do this is to use `\N` to mark that part of the string as non-expandable. For example:

  ```
  deny senders = \N^.*this.*@example\.com$\N : \
                 \N^\d{8}.+@spamhaus.example$\N : ...
  ```

  The `\N` sequences are removed by the expansion, so these items do indeed start with "^" by the time they are being interpreted as address patterns.

- Complete addresses can be looked up by using a pattern that starts with a lookup type terminated by a semicolon, followed by the data for the lookup. For example:

  ```
  deny senders = cdb;/etc/blocked.senders : \
    mysql;select address from blocked where \
    address='${quote_mysql:$sender_address}'
  ```

  Both query-style and single-key lookup types can be used. For a single-key lookup type, Exim uses the complete address as the key. However, empty keys are not supported for single-key lookups, so a match against the empty address always fails. This restriction does not apply to query-style lookups.

  Partial matching for single-key lookups (section 9.7) cannot be used, and is ignored if specified, with an entry being written to the panic log. However, you can configure lookup defaults, as described in section 9.6, but this is useful only for the "*@" type of default. For example, with this lookup:

  ```
  accept senders = lsearch*@;/some/file
  ```

  the file could contains lines like this:

  ```
  user1@domain1.example
  *@domain2.example
  ```

  and for the sender address *nimrod@jaeger.example*, the sequence of keys that are tried is:

  ```
  nimrod@jaeger.example
  *@jaeger.example
  *
  ```

Warning 1: Do not include a line keyed by "*" in the file, because that would mean that every address matches, thus rendering the test useless.

Warning 2: Do not confuse these two kinds of item:

```
deny recipients = dbm*@;/some/file
deny recipients = *@dbm;/some/file
```

The first does a whole address lookup, with defaulting, as just described, because it starts with a lookup type. The second matches the local part and domain independently, as described in a bullet point below.

The following kinds of address list pattern can match only non-empty addresses. If the subject address is empty, a match against any of these pattern types always fails.

- If a pattern starts with "@@" followed by a single-key lookup item (for example, `@@lsearch;/some/file`), the address that is 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 patterns, each of which is matched against the subject local part in turn.

  The lookup may be a partial one, and/or one involving a search for a default keyed by "*" (see section 9.6). The local part patterns that are looked up can be regular expressions or begin with "*", or even be further lookups. They may also be independently negated. For example, with

  ```
  deny senders = @@dbm;/etc/reject-by-domain
  ```

  the data from which the DBM file is built could contain lines like

  ```
  baddomain.com:  !postmaster : *
  ```

  to reject all senders except **postmaster** from that domain.

  If a local part that actually begins with an exclamation mark is required, it has to be specified using a regular expression. In *lsearch* files, an entry may be split over several lines by indenting the second and subsequent lines, but the separating colon must still be included at line breaks. White space surrounding the colons is ignored. For example:

  ```
  aol.com:  spammer1 : spammer2 : ^[0-9]+$ :
  spammer3 : spammer4
  ```

  As in all colon-separated lists in Exim, a colon can be included in an item by doubling.

  If the last item in the list 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. The new key can be any sequence of characters. Thus one might have entries like

  ```
  aol.com: spammer1 : spammer 2 : >*
  xyz.com: spammer3 : >*
  *:       ^\d{8}$
  ```

  in a file that was searched with **@@dbm***, to specify a match for 8-digit local parts for all domains, in addition to the specific local parts listed for each 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 chains may be no more than fifty items long.

- The @@<*lookup*> style of item can also be used with a query-style lookup, but in this case, the chaining facility is not available. The lookup can only return a single list of local parts.

Warning: There is an important difference between the address list items in these two examples:

```
senders = +my_list
senders = *@+my_list
```

In the first one, `my_list` is a named address list, whereas in the second example it is a named domain list.

10.20 Case of letters in address lists

Domains in email addresses are always handled caselessly, but for local parts case may be significant on some systems (see **caseful_local_part** for how Exim deals with this when routing addresses). However, RFC 2505 (*Anti-Spam Recommendations for SMTP MTAs*) suggests that matching of addresses to blocking lists should be done in a case-independent 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 lowercased before matching it to an address list. The local part is lowercased by default, and any string comparisons that take place are done caselessly. This means that the data in the address list itself, in files included as plain file names, and in any file that is looked up using the "@@" mechanism, can be in any case. However, the keys in files that are looked up by a search type other than *lsearch* (which works caselessly) must be in lower case, because these lookups are not case-independent.

To allow for the possibility of caseful address list matching, if an item in an address list is the string "+caseful", the original case of the local part is restored for any comparisons that follow, and string comparisons are no longer case-independent. This does not affect the domain, which remains in lower case. However, although independent matches on the domain alone are still performed caselessly, regular expressions that match against an entire address become case-sensitive after "+caseful" has been seen.

10.21 Local part lists

Case-sensitivity in local part lists is handled in the same way as for address lists, as just described. The "+caseful" item can be used if required. In a setting of the **local_parts** option in a router with **caseful_local_part** set false, the subject is lowercased and the matching is initially case-insensitive. In this case, "+caseful" will restore case-sensitive matching in the local part list, but not elsewhere in the router. If **caseful_local_part** is set true in a router, matching in the **local_parts** option is case-sensitive from the start.

If a local part list is indirected to a file (see section 10.3), comments are handled in the same way as address lists – they are recognized only if the # is preceded by white space or the start of the line. Otherwise, local part lists are matched in the same way as domain lists, except that the special items that refer to the local host (`@`, `@[]`, `@mx_any`, `@mx_primary`, and `@mx_secondary`) are not recognized. Refer to section 10.8 for details of the other available item types.

11. String expansions

Many strings in Exim's run time configuration are expanded before use. Some of them are expanded every time they are used; others are expanded only once.

When a string is being expanded it is copied verbatim from left to right except when a dollar or backslash character is encountered. A dollar specifies the start of a portion of the string that is interpreted and replaced as described below in section 11.5 onwards. Backslash is used as an escape character, as described in the following section.

11.1 Literal text in expanded strings

An uninterpreted dollar can be included in an expanded string by putting a backslash in front of it. A backslash can be used to prevent any special character being treated specially in an expansion, including backslash itself. If the string appears in quotes in the configuration file, two backslashes are required because the quotes themselves cause interpretation of backslashes when the string is read in (see section 6.16).

A portion of the string can specified as non-expandable by placing it between two occurrences of `\N`. This is particularly useful for protecting regular expressions, which often contain backslashes and dollar signs. For example:

```
deny senders = \N^\d{8}[a-z]@some\.site\.example$\N
```

On encountering the first `\N`, the expander copies subsequent characters without interpretation until it reaches the next `\N` or the end of the string.

11.2 Character escape sequences in expanded strings

A backslash followed by one of the letters "n", "r", or "t" in an expanded string is recognized as an escape sequence for the character newline, carriage return, or tab, respectively. A backslash followed by up to three octal digits is recognized as an octal encoding for a single character, and a backslash followed by "x" and up to two hexadecimal digits is a hexadecimal encoding.

These escape sequences are also recognized in quoted strings when they are read in. Their interpretation in expansions as well is useful for unquoted strings, and for other cases such as looked-up strings that are then expanded.

11.3 Testing string expansions

Many expansions can be tested by calling Exim with the **-be** option. This takes the command arguments, or lines from the standard input if there are no arguments, runs them through the string expansion code, and writes the results to the standard output. Variables based on configuration values are set up, but since no message is being processed, variables such as *$local_part* have no value. Nevertheless the **-be** option can be useful for checking out file and database lookups, and the use of expansion operators such as **sg**, **substr** and **nhash**.

Exim gives up its root privilege when it is called with the **-be** option, and instead runs under the uid and gid it was called with, to prevent users from using **-be** for reading files to which they do not have access.

If you want to test expansions that include variables whose values are taken from a message, there are two other options that can be used. The **-bem** option is like **-be** except that it is followed by a file name. The file is read as a message before doing the test expansions. For example:

```
exim -bem /tmp/test.message '$h_subject:'
```

The **-Mset** option is used in conjunction with **-be** and is followed by an Exim message identifier. For example:

```
exim -be -Mset 1GrA8W-0004WS-LQ '$recipients'
```

This loads the message from Exim's spool before doing the test expansions, and is therefore restricted to admin users.

11.4 Forced expansion failure

A number of expansions that are described in the following section have alternative "true" and "false" substrings, enclosed in brace characters (which are sometimes called "curly brackets"). Which of the two strings is used depends on some condition that is evaluated as part of the expansion. If, instead of a "false" substring, the word "fail" is used (not in braces), the entire string expansion fails in a way that can be detected by the code that requested the expansion. This is called "forced expansion failure", and its consequences depend on the circumstances. In some cases it is no different from any other expansion failure, but in others a different action may be taken. Such variations are mentioned in the documentation of the option that is being expanded.

11.5 Expansion items

The following items are recognized in expanded strings. White space may be used between sub-items that are keywords or substrings enclosed in braces inside an outer set of braces, to improve readability. **Warning**: Within braces, white space is significant.

$**<*variable name*> or **${<*variable name*>**}**
Substitute the contents of the named variable, for example:

```
$local_part
${domain}
```

The second form can be used to separate the name from subsequent alphanumeric characters. This form (using braces) is available only for variables; it does *not* apply to message headers. The names of the variables are given in section 11.9 below. If the name of a non-existent variable is given, the expansion fails.

${<*op*>**:**<*string*>**}**
The string is first itself expanded, and then the operation specified by <*op*> is applied to it. For example:

```
${lc:$local_part}
```

The string starts with the first character after the colon, which may be leading white space. A list of operators is given in section 11.6 below. The operator notation is used for simple expansion items that have just one argument, because it reduces the number of braces and therefore makes the string easier to understand.

$bheader_<*header name*>**:** or **$bh_**<*header name*>**:**
This item inserts "basic" header lines. It is described with the **header** expansion item below.

${dlfunc{<*file*>**}{**<*function*>**}{**<*arg*>**}{**<*arg*>**}...}**
This expansion dynamically loads and then calls a locally-written C function. This functionality is available only if Exim is compiled with

```
EXPAND_DLFUNC=yes
```

set in *Local/Makefile*. Once loaded, Exim remembers the dynamically loaded object so that it doesn't reload the same object file in the same Exim process (but of course Exim does start new processes frequently).

There may be from zero to eight arguments to the function. When compiling a local function that is to be called in this way, *local_scan.h* should be included. The Exim variables and functions that are defined by that API are also available for dynamically loaded functions. The function itself must have the following type:

```
int dlfunction(uschar **yield, int argc, uschar *argv[])
```

Where `uschar` is a typedef for `unsigned char` in *local_scan.h*. The function should return one of the following values:

`OK`: Success. The string that is placed in the variable *yield* is put into the expanded string that is being built.

`FAIL`: A non-forced expansion failure occurs, with the error message taken from *yield*, if it is set.

`FAIL_FORCED`: A forced expansion failure occurs, with the error message taken from *yield* if it is set.

`ERROR`: Same as `FAIL`, except that a panic log entry is written.

When compiling a function that is to be used in this way with gcc, you need to add **-shared** to the gcc command. Also, in the Exim build-time configuration, you must add **-export-dynamic** to EXTRALIBS.

${extract{<*key*>**}{**<*string1*>**}{**<*string2*>**}{**<*string3*>**}}**
The key and <*string1*> are first expanded separately. Leading and trailing white space is removed from the key (but not from any of the strings). The key must not consist entirely of digits. The expanded <*string1*> must be of the form:

<*key1*> = <*value1*> <*key2*> = <*value2*> ...

where the equals signs and spaces (but not both) are optional. If any of the values contain white space, they must be enclosed in double quotes, and any values that are enclosed in double quotes are subject to escape processing as described in section 6.16. The expanded <*string1*> is searched for the value that corresponds to the key. The search is case-insensitive. If the key is found, <*string2*> is expanded, and replaces the whole item; otherwise <*string3*> is used. During the expansion of <*string2*> the variable *$value* contains the value that has been extracted. Afterwards, it is restored to any previous value it might have had.

If {<*string3*>} is omitted, the item is replaced by an empty string if the key is not found. If {<*string2*>} is also omitted, the value that was extracted is used. Thus, for example, these two expansions are identical, and yield "2001":

```
${extract{gid}{uid=1984 gid=2001}}
${extract{gid}{uid=1984 gid=2001}{$value}}
```

Instead of {<*string3*>} the word "fail" (not in curly brackets) can appear, for example:

```
${extract{Z}{A=... B=...}{$value} fail }
```

This forces an expansion failure (see section 11.4); {<*string2*>} must be present for "fail" to be recognized.

${extract{<*number*>**}{**<*separators*>**}{**<*string1*>**}{**<*string2*>**}{**<*string3*>**}}**
The <*number*> argument must consist entirely of decimal digits, apart from leading and trailing white space, which is ignored. This is what distinguishes this form of **extract** from the previous kind. It behaves in the same way, except that, instead of extracting a named field, it extracts from <*string1*> the field whose number is given as the first argument. You can use *$value* in <*string2*> or `fail` instead of <*string3*> as before.

The fields in the string are separated by any one of the characters in the separator string. These may include space or tab characters. The first field is numbered one. If the number is negative, the fields are counted from the end of the string, with the rightmost one numbered -1. If the number given is zero, the entire string is returned. If the modulus of the number is greater than the number of fields in the string, the result is the expansion of <*string3*>, or the empty string if <*string3*> is not provided. For example:

```
${extract{2}{:}{x:42:99:& Mailer::/bin/bash}}
```

yields "42", and

```
${extract{-4}{:}{x:42:99:& Mailer::/bin/bash}}
```

yields "99". Two successive separators mean that the field between them is empty (for example, the fifth field above).

${filter{<*string*>**}{**<*condition*>**}}**
After expansion, <*string*> is interpreted as a list, colon-separated by default, but the separator can be changed in the usual way. For each item in this list, its value is place in *$item*, and then the condition is evaluated. If the condition is true, *$item* is added to the output as an item in a new list; if the condition is false, the item is discarded. The separator used for the output list is the same as the one used for the input, but a separator setting is not included in the output. For example:

```
${filter{a:b:c}{!eq{$item}{b}}
```

yields `a:c`. At the end of the expansion, the value of *$item* is restored to what it was before. See also the **map** and **reduce** expansion items.

${hash{<*string1*>**}{**<*string2*>**}{**<*string3*>**}}**
This is a textual hashing function, and was the first to be implemented in early versions of Exim. In current releases, there are other hashing functions (numeric, MD5, and SHA-1), which are described below.

The first two strings, after expansion, must be numbers. Call them <*m*> and <*n*>. If you are using fixed values for these numbers, that is, if <*string1*> and <*string2*> do not change when they are expanded, you can use the simpler operator notation that avoids some of the braces:

```
${hash_<n>_<m>:<string>}
```

The second number is optional (in both notations). If <*n*> is greater than or equal to the length of the string, the expansion item returns the string. Otherwise it computes a new string of length <*n*> by applying a hashing function to the string. The new string consists of characters taken from the first <*m*> characters of the string

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQWRSTUVWXYZ0123456789
```

If <*m*> is not present the value 26 is used, so that only lower case letters appear. For example:

```
$hash{3}{monty}}             yields jmg
$hash{5}{monty}}             yields monty
$hash{4}{62}{monty python}}  yields fbWx
```

$header_<*header name*>**:** or **$h_**<*header name*>**:**
$bheader_<*header name*>**:** or **$bh_**<*header name*>**:**
$rheader_<*header name*>**:** or **$rh_**<*header name*>**:**
Substitute the contents of the named message header line, for example

```
$header_reply-to:
```

The newline that terminates a header line is not included in the expansion, but internal newlines (caused by splitting the header line over several physical lines) may be present.

The difference between **rheader**, **bheader**, and **header** is in the way the data in the header line is interpreted.

- **rheader** gives the original "raw" content of the header line, with no processing at all, and without the removal of leading and trailing white space.
- **bheader** removes leading and trailing white space, and then decodes base64 or quoted-printable MIME "words" within the header text, but does no character set translation. If decoding of what looks superficially like a MIME "word" fails, the raw string is returned. If decoding produces a binary zero character, it is replaced by a question mark – this is what Exim does for binary zeros that are actually received in header lines.
- **header** tries to translate the string as decoded by **bheader** to a standard character set. This is an attempt to produce the same string as would be displayed on a user's MUA. If translation fails, the **bheader** string is returned. Translation is attempted only on operating systems that support the *iconv()* function. This is indicated by the compile-time macro HAVE_ICONV in a system Makefile or in *Local/Makefile*.

In a filter file, the target character set for **header** can be specified by a command of the following form:

```
headers charset "UTF-8"
```

This command affects all references to *$h_* (or *$header_*) expansions in subsequently obeyed filter commands. In the absence of this command, the target character set in a filter is taken from the setting of the **headers_charset** option in the runtime configuration. The value of this option defaults to the value of HEADERS_CHARSET in *Local/Makefile*. The ultimate default is ISO-8859-1.

Header names follow the syntax of RFC 2822, which states that they may contain any printing characters except space and colon. Consequently, curly brackets *do not* terminate header names, and should not be used to enclose them as if they were variables. Attempting to do so causes a syntax error.

Only header lines that are common to all copies of a message are visible to this mechanism. These are the original header lines that are received with the message, and any that are added by an ACL statement or by a system filter. Header lines that are added to a particular copy of a message by a router or transport are not accessible.

For incoming SMTP messages, no header lines are visible in ACLs that are obeyed before the DATA ACL, because the header structure is not set up until the message is received. Header lines that are added in a RCPT ACL (for example) are saved until the message's incoming header lines are available, at which point they are added. When a DATA ACL is running, however, header lines added by earlier ACLs are visible.

Upper case and lower case letters are synonymous in header names. If the following character is white space, the terminating colon may be omitted, but this is not recommended, because you may then forget it when it is needed. When white space terminates the header name, it is included in the expanded string. If the message does not contain the given header, the expansion item is replaced by an empty string. (See the **def** condition in section 11.7 for a means of testing for the existence of a header.)

If there is more than one header with the same name, they are all concatenated to form the substitution string, up to a maximum length of 64K. Unless **rheader** is being used, leading and trailing white space is removed from each header before concatenation, and a completely empty header is ignored. A newline character is then inserted between non-empty headers, but there is no newline at the very end. For the **header** and **bheader** expansion, for those headers that contain lists of addresses, a comma is also inserted at the junctions between headers. This does not happen for the **rheader** expansion.

${hmac{<*hashname*>**}{**<*secret*>**}{**<*string*>**}}**

This function uses cryptographic hashing (either MD5 or SHA-1) to convert a shared secret and some text into a message authentication code, as specified in RFC 2104. This differs from `${md5:secret_text...}` or `${sha1:secret_text...}` in that the hmac step adds a signature to the cryptographic hash, allowing for authentication that is not possible with MD5 or SHA-1 alone. The hash name must expand to either `md5` or `sha1` at present. For example:

```
${hmac{md5}{somesecret}{$primary_hostname $tod_log}}
```

For the hostname *mail.example.com* and time 2002-10-17 11:30:59, this produces:

```
dd97e3ba5d1a61b5006108f8c8252953
```

As an example of how this might be used, you might put in the main part of an Exim configuration:

```
SPAMSCAN_SECRET=cohgheeLei2thahw
```

In a router or a transport you could then have:

```
headers_add = \
  X-Spam-Scanned: ${primary_hostname} ${message_exim_id} \
  ${hmac{md5}{SPAMSCAN_SECRET}\
  {${primary_hostname},${message_exim_id},$h_message-id:}}
```

Then given a message, you can check where it was scanned by looking at the *X-Spam-Scanned:* header line. If you know the secret, you can check that this header line is authentic by recomputing the authentication code from the host name, message ID and the *Message-id:* header line. This can be done using Exim's **-be** option, or by other means, for example by using the *hmac_md5_hex()* function in Perl.

${if <*condition*> **{**<*string1*>**}{**<*string2*>**}}**
If <*condition*> is true, <*string1*> is expanded and replaces the whole item; otherwise <*string2*> is used. The available conditions are described in section 11.7 below. For example:

```
${if eq {$local_part}{postmaster} {yes}{no} }
```

The second string need not be present; if it is not and the condition is not true, the item is replaced with nothing. Alternatively, the word "fail" may be present instead of the second string (without any curly brackets). In this case, the expansion is forced to fail if the condition is not true (see section 11.4).

If both strings are omitted, the result is the string `true` if the condition is true, and the empty string if the condition is false. This makes it less cumbersome to write custom ACL and router conditions. For example, instead of

```
condition = ${if >{$acl_m4}{3}{true}{false}}
```

you can use

```
condition = ${if >{$acl_m4}{3}}
```

${length{<*string1*>**}{**<*string2*>**}}**
The **length** item is used to extract the initial portion of a string. Both strings are expanded, and the first one must yield a number, <*n*>, say. If you are using a fixed value for the number, that is, if <*string1*> does not change when expanded, you can use the simpler operator notation that avoids some of the braces:

```
${length_<n>:<string>}
```

The result of this item is either the first <*n*> characters or the whole of <*string2*>, whichever is the shorter. Do not confuse **length** with **strlen**, which gives the length of a string.

${lookup{<*key*>**}** <*search type*> **{**<*file*>**} {**<*string1*>**} {**<*string2*>**}}**
This is the first of one of two different types of lookup item, which are both described in the next item.

${lookup <*search type*> **{**<*query*>**} {**<*string1*>**} {**<*string2*>**}}**
The two forms of lookup item specify data lookups in files and databases, as discussed in chapter 9. The first form is used for single-key lookups, and the second is used for query-style lookups. The <*key*>, <*file*>, and <*query*> strings are expanded before use.

If there is any white space in a lookup item which is part of a filter command, a retry or rewrite rule, a routing rule for the *manualroute* router, or any other place where white space is significant, the lookup item must be enclosed in double quotes. The use of data lookups in users' filter files may be locked out by the system administrator.

If the lookup succeeds, <*string1*> is expanded and replaces the entire item. During its expansion, the variable *$value* contains the data returned by the lookup. Afterwards it reverts to the value it had previously (at the outer level it is empty). If the lookup fails, <*string2*> is expanded and replaces the entire item. If {<*string2*>} is omitted, the replacement is the empty string on failure. If <*string2*> is provided, it can itself be a nested lookup, thus providing a mechanism for looking up a default value when the original lookup fails.

If a nested lookup is used as part of <*string1*>, *$value* contains the data for the outer lookup while the parameters of the second lookup are expanded, and also while <*string2*> of the second lookup is expanded, should the second lookup fail. Instead of {<*string2*>} the word "fail" can appear, and in this case, if the lookup fails, the entire expansion is forced to fail (see section 11.4). If both {<*string1*>} and {<*string2*>} are omitted, the result is the looked up value in the case of a successful lookup, and nothing in the case of failure.

For single-key lookups, the string "partial" is permitted to precede the search type in order to do partial matching, and * or *@ may follow a search type to request default lookups if the key does not match (see sections 9.6 and 9.7 for details).

If a partial search is used, the variables *$1* and *$2* contain the wild and non-wild parts of the key during the expansion of the replacement text. They return to their previous values at the end of the lookup item.

This example looks up the postmaster alias in the conventional alias file:

```
${lookup {postmaster} lsearch {/etc/aliases} {$value}}
```

This example uses NIS+ to look up the full name of the user corresponding to the local part of an address, forcing the expansion to fail if it is not found:

```
${lookup nisplus {[name=$local_part],passwd.org_dir:gcos} \
  {$value}fail}
```

${map{<*string1*>**}{**<*string2*>**}}**
After expansion, <*string1*> is interpreted as a list, colon-separated by default, but the separator can be changed in the usual way. For each item in this list, its value is place in *$item*, and then <*string2*> is expanded and added to the output as an item in a new list. The separator used for the output list is the same as the one used for the input, but a separator setting is not included in the output. For example:

```
${map{a:b:c}{[$item]}} ${map{<- x-y-z}{($item)}}
```

expands to `[a]:[b]:[c] (x)-(y)-(z)`. At the end of the expansion, the value of *$item* is restored to what it was before. See also the **filter** and **reduce** expansion items.

${nhash{<*string1*>**}{**<*string2*>**}{**<*string3*>**}}**
The three strings are expanded; the first two must yield numbers. Call them <*n*> and <*m*>. If you are using fixed values for these numbers, that is, if <*string1*> and <*string2*> do not change when they are expanded, you can use the simpler operator notation that avoids some of the braces:

```
${nhash_<n>_<m>:<string>}
```

The second number is optional (in both notations). If there is only one number, the result is a number in the range 0–<*n*>-1. Otherwise, the string is 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. For example,

```
${nhash{8}{64}{supercalifragilisticexpialidocious}}
```

returns the string "6/33".

${perl{<*subroutine*>**}{**<*arg*>**}{**<*arg*>**}...}**
This item is available only if Exim has been built to include an embedded Perl interpreter. The subroutine name and the arguments are first separately expanded, and then the Perl subroutine is called with those arguments. No additional arguments need be given; the maximum number permitted, including the name of the subroutine, is nine.

The return value of the subroutine is inserted into the expanded string, unless the return value is **undef**. In that case, the expansion fails in the same way as an explicit "fail" on a lookup item. The return value is a scalar. Whatever you return is evaluated in a scalar context. For example, if you return the name of a Perl vector, the return value is the size of the vector, not its contents.

If the subroutine exits by calling Perl's **die** function, the expansion fails with the error message that was passed to **die**. More details of the embedded Perl facility are given in chapter 12.

The *redirect* router has an option called **forbid_filter_perl** which locks out the use of this expansion item in filter files.

${prvs{<*address*>**}{**<*secret*>**}{**<*keynumber*>**}}**
The first argument is a complete email address and the second is secret keystring. The third argument, specifying a key number, is optional. If absent, it defaults to 0. The result of the expansion is a prvs-signed email address, to be typically used with the **return_path** option on an

smtp transport as part of a bounce address tag validation (BATV) scheme. For more discussion and an example, see section 40.47.

${prvscheck{<*address*>**}{**<*secret*>**}{**<*string*>**}}**
This expansion item is the complement of the **prvs** item. It is used for checking prvs-signed addresses. If the expansion of the first argument does not yield a syntactically valid prvs-signed address, the whole item expands to the empty string. When the first argument does expand to a syntactically valid prvs-signed address, the second argument is expanded, with the prvs-decoded version of the address and the key number extracted from the address in the variables *$prvscheck_address* and *$prvscheck_keynum*, respectively.

These two variables can be used in the expansion of the second argument to retrieve the secret. The validity of the prvs-signed address is then checked against the secret. The result is stored in the variable *$prvscheck_result*, which is empty for failure or "1" for success.

The third argument is optional; if it is missing, it defaults to an empty string. This argument is now expanded. If the result is an empty string, the result of the expansion is the decoded version of the address. This is the case whether or not the signature was valid. Otherwise, the result of the expansion is the expansion of the third argument.

All three variables can be used in the expansion of the third argument. However, once the expansion is complete, only *$prvscheck_result* remains set. For more discussion and an example, see section 40.47.

${readfile{<*file name*>**}{**<*eol string*>**}}**
The file name and end-of-line string are first expanded separately. The file is then read, and its contents replace the entire item. All newline characters in the file are replaced by the end-of-line string if it is present. Otherwise, newlines are left in the string. String expansion is not applied to the contents of the file. If you want this, you must wrap the item in an **expand** operator. If the file cannot be read, the string expansion fails.

The *redirect* router has an option called **forbid_filter_readfile** which locks out the use of this expansion item in filter files.

${readsocket{<*name*>**}{**<*request*>**}{**<*timeout*>**}{**<*eol string*>**}{**<*fail string*>**}}**
This item inserts data from a Unix domain or Internet socket into the expanded string. The minimal way of using it uses just two arguments, as in these examples:

```
${readsocket{/socket/name}{request string}}
${readsocket{inet:some.host:1234}{request string}}
```

For a Unix domain socket, the first substring must be the path to the socket. For an Internet socket, the first substring must contain `inet:` followed by a host name or IP address, followed by a colon and a port, which can be a number or the name of a TCP port in */etc/services*. An IP address may optionally be enclosed in square brackets. This is best for IPv6 addresses. For example:

```
${readsocket{inet:[::1]:1234}{request string}}
```

Only a single host name may be given, but if looking it up yields more than one IP address, they are each tried in turn until a connection is made. For both kinds of socket, Exim makes a connection, writes the request string (unless it is an empty string) and reads from the socket until an end-of-file is read. A timeout of 5 seconds is applied. Additional, optional arguments extend what can be done. Firstly, you can vary the timeout. For example:

```
${readsocket{/socket/name}{request string}{3s}}
```

A fourth argument allows you to change any newlines that are in the data that is read, in the same way as for **readfile** (see above). This example turns them into spaces:

```
${readsocket{inet:127.0.0.1:3294}{request string}{3s}{ }}
```

As with all expansions, the substrings are expanded before the processing happens. Errors in these sub-expansions cause the expansion to fail. In addition, the following errors can occur:

- Failure to create a socket file descriptor;

- Failure to connect the socket;
- Failure to write the request string;
- Timeout on reading from the socket.

By default, any of these errors causes the expansion to fail. However, if you supply a fifth substring, it is expanded and used when any of the above errors occurs. For example:

```
${readsocket{/socket/name}{request string}{3s}{\n}\
  {socket failure}}
```

You can test for the existence of a Unix domain socket by wrapping this expansion in `${if exists`, but there is a race condition between that test and the actual opening of the socket, so it is safer to use the fifth argument if you want to be absolutely sure of avoiding an expansion error for a non-existent Unix domain socket, or a failure to connect to an Internet socket.

The *redirect* router has an option called **forbid_filter_readsocket** which locks out the use of this expansion item in filter files.

${reduce{<*string1*>**}{**<*string2*>**}{**<*string3*>**}}**
This operation reduces a list to a single, scalar string. After expansion, <*string1*> is interpreted as a list, colon-separated by default, but the separator can be changed in the usual way. Then <*string2*> is expanded and assigned to the *$value* variable. After this, each item in the <*string1*> list is assigned to *$item* in turn, and <*string3*> is expanded for each of them. The result of that expansion is assigned to *$value* before the next iteration. When the end of the list is reached, the final value of *$value* is added to the expansion output. The **reduce** expansion item can be used in a number of ways. For example, to add up a list of numbers:

```
${reduce {<, 1,2,3}{0}{${eval:$value+$item}}}
```

The result of that expansion would be `6`. The maximum of a list of numbers can be found:

```
${reduce {3:0:9:4:6}{0}{${if >{$item}{$value}{$item}{$value}}}}
```

At the end of a **reduce** expansion, the values of *$item* and *$value* are restored to what they were before. See also the **filter** and **map** expansion items.

$rheader_<*header name*>**:** or **$rh_**<*header name*>**:**
This item inserts "raw" header lines. It is described with the **header** expansion item above.

${run{<*command*> <*args*>**}{**<*string1*>**}{**<*string2*>**}}**
The command and its arguments are first expanded separately, and then the command is run in a separate process, but under the same uid and gid. As in other command executions from Exim, a shell is not used by default. If you want a shell, you must explicitly code it.

The standard input for the command exists, but is empty. The standard output and standard error are set to the same file descriptor. If the command succeeds (gives a zero return code) <*string1*> is expanded and replaces the entire item; during this expansion, the standard output/error from the command is in the variable *$value*. If the command fails, <*string2*>, if present, is expanded and used. Once again, during the expansion, the standard output/error from the command is in the variable *$value*.

If <*string2*> is absent, the result is empty. Alternatively, <*string2*> can be the word "fail" (not in braces) to force expansion failure if the command does not succeed. If both strings are omitted, the result is contents of the standard output/error on success, and nothing on failure.

The return code from the command is put in the variable *$runrc*, and this remains set afterwards, so in a filter file you can do things like this:

```
if "${run{x y z}{}}$runrc" is 1 then ...
  elif $runrc is 2 then ...
  ...
endif
```

If execution of the command fails (for example, the command does not exist), the return code is 127 – the same code that shells use for non-existent commands.

Warning: In a router or transport, you cannot assume the order in which option values are expanded, except for those preconditions whose order of testing is documented. Therefore, you cannot reliably expect to set *$runrc* by the expansion of one option, and use it in another.

The *redirect* router has an option called **forbid_filter_run** which locks out the use of this expansion item in filter files.

${sg{<*subject*>**}{**<*regex*>**}{**<*replacement*>**}}**
This item works like Perl's substitution operator (s) with the global (/g) option; hence its name. However, unlike the Perl equivalent, Exim does not modify the subject string; instead it returns the modified string for insertion into the overall expansion. The item takes three arguments: the subject string, a regular expression, and a substitution string. For example:

```
${sg{abcdefabcdef}{abc}{xyz}}
```

yields "xyzdefxyzdef". Because all three arguments are expanded before use, if any $ or \ characters are required in the regular expression or in the substitution string, they have to be escaped. For example:

```
${sg{abcdef}{^(...)(...)\$}{\$2\$1}}
```

yields "defabc", and

```
${sg{1=A 4=D 3=C}{\N(\d+)=\N}{K\$1=}}
```

yields "K1=A K4=D K3=C". Note the use of `\N` to protect the contents of the regular expression from string expansion.

${substr{<*string1*>**}{**<*string2*>**}{**<*string3*>**}}**
The three strings are expanded; the first two must yield numbers. Call them <*n*> and <*m*>. If you are using fixed values for these numbers, that is, if <*string1*> and <*string2*> do not change when they are expanded, you can use the simpler operator notation that avoids some of the braces:

```
${substr_<n>_<m>:<string>}
```

The second number is optional (in both notations). If it is absent in the simpler format, the preceding underscore must also be omitted.

The **substr** item can be used to extract more general substrings than **length**. The first number, <*n*>, is a starting offset, and <*m*> is the length required. For example

```
${substr{3}{2}{$local_part}}
```

If the starting offset is greater than the string length the result is the null string; if the length plus starting offset is greater than the string length, the result is the right-hand part of the string, starting from the given offset. The first character in the string has offset zero.

The **substr** expansion item can take negative offset values to count from the right-hand end of its operand. The last character is offset -1, the second-last is offset -2, and so on. Thus, 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. Thus, for example,

```
${substr{-5}{2}{12}}
```

yields an empty string, but

```
${substr{-3}{2}{12}}
```

yields "1".

When 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, an offset of -1 and no length, as in these semantically identical examples:

```
${substr_-1:abcde}
${substr{-1}{abcde}}
```

yields all but the last character of the string, that is, “abcd”.

${tr{<*subject*>**}{**<*characters*>**}{**<*replacements*>**}}**
This item does single-character translation on its subject string. The second argument is a list of characters to be translated in the subject string. Each matching character is replaced by the corresponding character from the replacement list. For example

```
${tr{abcdea}{ac}{13}}
```

yields `1b3de1`. If there are duplicates in the second character 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.

11.6 Expansion operators

For expansion items that perform transformations on a single argument string, the “operator” notation is used because it is simpler and uses fewer braces. The substring is first expanded before the operation is applied to it. The following operations can be performed:

${address:<*string*>**}**
The string is interpreted as an RFC 2822 address, as it might appear in a header line, and the effective address is extracted from it. If the string does not parse successfully, the result is empty.

${addresses:<*string*>**}**
The string (after expansion) is interpreted as a list of addresses in RFC 2822 format, such as can be found in a *To:* or *Cc:* header line. The operative address (*local-part@domain*) is extracted from each item, and the result of the expansion is a colon-separated list, with appropriate doubling of colons should any happen to be present in the email addresses. Syntactically invalid RFC2822 address items are omitted from the output.

It is possible to specify a character other than colon for the output separator by starting the string with > followed by the new separator character. For example:

```
${addresses:>& Chief <ceo@up.stairs>, sec@base.ment (dogsbody)}
```

expands to `ceo@up.stairs&sec@base.ment`. Compare the **address** (singular) expansion item, which extracts the working address from a single RFC2822 address. See the **filter**, **map**, and **reduce** items for ways of processing lists.

${base62:<*digits*>**}**
The string must consist entirely of decimal digits. The number is converted to base 62 and output as a string of six characters, including leading zeros. In the few operating environments where Exim uses base 36 instead of base 62 for its message identifiers (because those systems do not have case-sensitive file names), base 36 is used by this operator, despite its name. **Note**: Just to be absolutely clear: this is *not* base64 encoding.

${base62d:<*base-62 digits*>**}**
The string must consist entirely of base-62 digits, or, in operating environments where Exim uses base 36 instead of base 62 for its message identifiers, base-36 digits. The number is converted to decimal and output as a string.

${domain:<*string*>**}**
The string is interpreted as an RFC 2822 address and the domain is extracted from it. If the string does not parse successfully, the result is empty.

${escape:<*string*>**}**
If the string contains any non-printing characters, they are converted to escape sequences starting with a backslash. Whether characters with the most significant bit set (so-called “8-bit characters”) count as printing or not is controlled by the **print_topbitchars** option.

${eval:**<*string*>**}** and **${eval10:<*string*>**}**

These items supports simple arithmetic and bitwise logical operations in expansion strings. The string (after expansion) must be a conventional arithmetic expression, but it is limited to basic arithmetic operators, bitwise logical operators, and parentheses. All operations are carried out using integer arithmetic. The operator priorities are as follows (the same as in the C programming language):

highest:	not (~), negate (-)
	multiply (*), divide (/), remainder (%)
	plus (+), minus (-)
	shift-left (<<), shift-right (>>)
	and (&)
	xor (^)
lowest:	or (|)

Binary operators with the same priority are evaluated from left to right. White space is permitted before or after operators.

For **eval**, numbers may be decimal, octal (starting with "0") or hexadecimal (starting with "0x"). For **eval10**, all numbers are taken as decimal, even if they start with a leading zero; hexadecimal numbers are not permitted. This can be useful when processing numbers extracted from dates or times, which often do have leading zeros.

A number may be followed by "K" or "M" to multiply it by 1024 or 1024*1024, respectively. Negative numbers are supported. The result of the computation is a decimal representation of the answer (without "K" or "M"). For example:

```
${eval:1+1}             yields 2
${eval:1+2*3}           yields 7
${eval:(1+2)*3}         yields 9
${eval:2+42%5}          yields 4
${eval:0xc&5}           yields 4
${eval:0xc|5}           yields 13
${eval:0xc^5}           yields 9
${eval:0xc>>1}          yields 6
${eval:0xc<<1}          yields 24
${eval:~255&0x1234}     yields 4608
${eval:-(~255&0x1234)}  yields -4608
```

As a more realistic example, in an ACL you might have

```
deny   message = Too many bad recipients
       condition =                      \
         ${if and {                     \
           {>{$rcpt_count}{10}}         \
           {                            \
           <                            \
             {$recipients_count}        \
             {${eval:$rcpt_count/2}}    \
           }                            \
         }{yes}{no}}
```

The condition is true if there have been more than 10 RCPT commands and fewer than half of them have resulted in a valid recipient.

${expand:<*string*>**}**

The **expand** operator causes a string to be expanded for a second time. For example,

```
${expand:${lookup{$domain}dbm{/some/file}{$value}}}
```

first looks up a string in a file while expanding the operand for **expand**, and then re-expands what it has found.

${from_utf8:<*string*>**}**
The world is slowly moving towards Unicode, although there are no standards for email yet. However, other applications (including some databases) are starting to store data in Unicode, using UTF-8 encoding. This operator converts from a UTF-8 string to an ISO-8859-1 string. UTF-8 code values greater than 255 are converted to underscores. The input must be a valid UTF-8 string. If it is not, the result is an undefined sequence of bytes.

Unicode code points with values less than 256 are compatible with ASCII and ISO-8859-1 (also known as Latin-1). For example, character 169 is the copyright symbol in both cases, though the way it is encoded is different. In UTF-8, more than one byte is needed for characters with code values greater than 127, whereas ISO-8859-1 is a single-byte encoding (but thereby limited to 256 characters). This makes translation from UTF-8 to ISO-8859-1 straightforward.

${hash_<*n*>**_**<*m*>**:**<*string*>**}**
The **hash** operator is a simpler interface to the hashing function that can be used when the two parameters are fixed numbers (as opposed to strings that change when expanded). The effect is the same as

```
${hash{<n>}{<m>}{<string>}}
```

See the description of the general **hash** item above for details. The abbreviation **h** can be used when **hash** is used as an operator.

${hex2b64:<*hexstring*>**}**
This operator converts a hex string into one that is base64 encoded. This can be useful for processing the output of the MD5 and SHA-1 hashing functions.

${lc:<*string*>**}**
This forces the letters in the string into lower-case, for example:

```
${lc:$local_part}
```

${length_<*number*>**:**<*string*>**}**
The **length** operator is a simpler interface to the **length** function that can be used when the parameter is a fixed number (as opposed to a string that changes when expanded). The effect is the same as

```
${length{<number>}{<string>}}
```

See the description of the general **length** item above for details. Note that **length** is not the same as **strlen**. The abbreviation **l** can be used when **length** is used as an operator.

${local_part:<*string*>**}**
The string is interpreted as an RFC 2822 address and the local part is extracted from it. If the string does not parse successfully, the result is empty.

${mask:<*IP address*>**/**<*bit count*>**}**
If the form of the string to be operated on is not an IP address followed by a slash and an integer (that is, a network address in CIDR notation), the expansion fails. Otherwise, this operator converts the IP address to binary, masks off the least significant bits according to the bit count, and converts the result back to text, with mask appended. For example,

```
${mask:10.111.131.206/28}
```

returns the string "10.111.131.192/28". Since this operation is expected to be mostly used for looking up masked addresses in files, the result for an IPv6 address uses dots to separate components instead of colons, because colon terminates a key string in lsearch files. So, for example,

```
${mask:3ffe:ffff:836f:0a00:000a:0800:200a:c031/99}
```

returns the string

```
3ffe.ffff.836f.0a00.000a.0800.2000.0000/99
```

Letters in IPv6 addresses are always output in lower case.

${md5:<*string*>**}**
The **md5** operator computes the MD5 hash value of the string, and returns it as a 32-digit hexadecimal number, in which any letters are in lower case.

${nhash_<*n*>**_**<*m*>**:**<*string*>**}**
The **nhash** operator is a simpler interface to the numeric hashing function that can be used when the two parameters are fixed numbers (as opposed to strings that change when expanded). The effect is the same as

```
${nhash{<n>}{<m>}{<string>}}
```

See the description of the general **nhash** item above for details.

${quote:<*string*>**}**
The **quote** operator puts its argument into double quotes if it is an empty string or contains anything other than letters, digits, underscores, dots, and hyphens. Any occurrences of double quotes and backslashes are escaped with a backslash. Newlines and carriage returns are converted to `\n` and `\r`, respectively For example,

```
${quote:ab"*"cd}
```

becomes

```
"ab\"*\"cd"
```

The place where this is useful is when the argument is a substitution from a variable or a message header.

${quote_local_part:<*string*>**}**
This operator is like **quote**, except that it quotes the string only if required to do so by the rules of RFC 2822 for quoting local parts. For example, a plus sign would not cause quoting (but it would for **quote**). If you are creating a new email address from the contents of *$local_part* (or any other unknown data), you should always use this operator.

${quote_<*lookup-type*>**:**<*string*>**}**
This operator applies lookup-specific quoting rules to the string. Each query-style lookup type has its own quoting rules which are described with the lookups in chapter 9. For example,

```
${quote_ldap:two * two}
```

returns

```
two%20%5C2A%20two
```

For single-key lookup types, no quoting is ever necessary and this operator yields an unchanged string.

${rfc2047:<*string*>**}**
This operator encodes text according to the rules of RFC 2047. This is an encoding that is used in header lines to encode non-ASCII characters. It is assumed that the input string is in the encoding specified by the **headers_charset** option, which defaults to ISO-8859-1. If the string contains only characters in the range 33–126, and no instances of the characters

```
? = ( ) < > @ , ; : \ " . [ ] _
```

it is not modified. Otherwise, the result is the RFC 2047 encoding of the string, using as many "encoded words" as necessary to encode all the characters.

${rfc2047d:<*string*>**}**
This operator decodes strings that are encoded as per RFC 2047. Binary zero bytes are replaced by question marks. Characters are converted into the character set defined by **headers_charset**. Overlong RFC 2047 "words" are not recognized unless **check_rfc2047_length** is set false.

Note: If you use **$header_***xxx***:** (or **$h_***xxx***:**) to access a header line, RFC 2047 decoding is done automatically. You do not need to use this operator as well.

${rxquote:<*string*>**}**
The **rxquote** operator inserts a backslash before any non-alphanumeric characters in its argument. This is useful when substituting the values of variables or headers inside regular expressions.

${sha1:<*string*>**}**
The **sha1** operator computes the SHA-1 hash value of the string, and returns it as a 40-digit hexadecimal number, in which any letters are in upper case.

${stat:<*string*>**}**
The string, after expansion, must be a file path. A call to the *stat()* function is made for this path. If *stat()* fails, an error occurs and the expansion fails. If it succeeds, the data from the stat replaces the item, as a series of <*name*>=<*value*> pairs, where the values are all numerical, except for the value of "smode". The names are: "mode" (giving the mode as a 4-digit octal number), "smode" (giving the mode in symbolic format as a 10-character string, as for the *ls* command), "inode", "device", "links", "uid", "gid", "size", "atime", "mtime", and "ctime". You can extract individual fields using the **extract** expansion item.

The use of the **stat** expansion in users' filter files can be locked out by the system administrator. **Warning**: The file size may be incorrect on 32-bit systems for files larger than 2GB.

${str2b64:<*string*>**}**
This operator converts a string into one that is base64 encoded.

${strlen:<*string*>**}**
The item is replace by the length of the expanded string, expressed as a decimal number. **Note**: Do not confuse **strlen** with **length**.

${substr_<*start*>**_**<*length*>**:**<*string*>**}**
The **substr** operator is a simpler interface to the **substr** function that can be used when the two parameters are fixed numbers (as opposed to strings that change when expanded). The effect is the same as

```
${substr{<start>}{<length>}{<string>}}
```

See the description of the general **substr** item above for details. The abbreviation **s** can be used when **substr** is used as an operator.

${time_eval:<*string*>**}**
This item converts an Exim time interval such as `2d4h5m` into a number of seconds.

${time_interval:<*string*>**}**
The argument (after sub-expansion) must be a sequence of decimal digits that represents an interval of time as a number of seconds. It is converted into a number of larger units and output in Exim's normal time format, for example, `1w3d4h2m6s`.

${uc:<*string*>**}**
This forces the letters in the string into upper-case.

11.7 Expansion conditions

The following conditions are available for testing by the **${if** construct 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
==	equal
>	greater
>=	greater or equal
<	less
<=	less or equal

For example:

```
${if >{$message_size}{10M} ...
```

Note that the general negation operator provides for inequality testing. 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 lower case), signifying multiplication by 1024 or 1024*1024, respectively. As a special case, the numerical value of an empty string is taken as zero.

crypteq {<*string1*>**}{**<*string2*>**}**
This condition is included in the Exim binary if it is built to support any authentication mechanisms (see chapter 33). Otherwise, it is necessary to define SUPPORT_CRYPTEQ in *Local/Makefile* to get **crypteq** included in the binary.

The **crypteq** condition has two arguments. The first is encrypted and compared against the second, which is already encrypted. The second string may be in the LDAP form for storing encrypted strings, which starts with the encryption type in curly brackets, followed by the data. If the second string does not begin with "{" it is assumed to be encrypted with *crypt()* or *crypt16()* (see below), since such strings cannot begin with "{". Typically this will be a field from a password file. An example of an encrypted string in LDAP form is:

```
{md5}CY9rzUYh03PK3k6DJie09g==
```

If such a string appears directly in an expansion, the curly brackets have to be quoted, because they are part of the expansion syntax. For example:

```
${if crypteq {test}{\{md5\}CY9rzUYh03PK3k6DJie09g==}{yes}{no}}
```

The following encryption types (whose names are matched case-independently) are supported:

- **{md5}** computes the MD5 digest of the first string, and expresses this as printable characters to compare with the remainder of the second string. If the length of the comparison string is 24, Exim assumes that it is base64 encoded (as in the above example). If the length is 32, Exim assumes that it is a hexadecimal encoding of the MD5 digest. If the length not 24 or 32, the comparison fails.
- **{sha1}** computes the SHA-1 digest of the first string, and expresses this as printable characters to compare with the remainder of the second string. If the length of the comparison string is 28, Exim assumes that it is base64 encoded. If the length is 40, Exim assumes that it is a hexadecimal encoding of the SHA-1 digest. If the length is not 28 or 40, the comparison fails.
- **{crypt}** calls the *crypt()* function, which traditionally used to use only the first eight characters of the password. However, in modern operating systems this is no longer true, and in many cases the entire password is used, whatever its length.
- **{crypt16}** calls the *crypt16()* function, which was originally created to use up to 16 characters of the password in some operating systems. Again, in modern operating systems, more characters may be used.

Exim has its own version of *crypt16()*, which is just a double call to *crypt()*. For operating systems that have their own version, setting HAVE_CRYPT16 in *Local/Makefile* when building Exim causes it to use the operating system version instead of its own. This option is set by default in the OS-dependent *Makefile* for those operating systems that are known to support *crypt16()*.

Some years after Exim's *crypt16()* was implemented, a user discovered that it was not using the same algorithm as some operating systems' versions. It turns out that as well as *crypt16()* there is a function called *bigcrypt()* in some operating systems. This may or may not use the same algorithm, and both of them may be different to Exim's built-in *crypt16()*.

However, since there is now a move away from the traditional *crypt()* functions towards using SHA1 and other algorithms, tidying up this area of Exim is seen as very low priority.

If you do not put a encryption type (in curly brackets) in a **crypteq** comparison, the default is usually either `{crypt}` or `{crypt16}`, as determined by the setting of DEFAULT_CRYPT in *Local/Makefile*. The default default is `{crypt}`. Whatever the default, you can always use either function by specifying it explicitly in curly brackets.

def:<*variable name*>
The **def** condition must be followed by the name of one of the expansion variables defined in section 11.9. The condition is true if the variable does not contain the empty string. For example:

```
${if def:sender_ident {from $sender_ident}}
```

Note that the variable name is given without a leading **$** character. If the variable does not exist, the expansion fails.

def:header_<*header name*>**:** or **def:h_**<*header name*>**:**
This condition is true if a message is being processed and the named header exists in the message. For example,

```
${if def:header_reply-to:{$h_reply-to:}{$h_from:}}
```

Note: No **$** appears before **header_** or **h_** in the condition, and the header name must be terminated by a colon if white space does not follow.

eq {<*string1*>**}{**<*string2*>**}**
eqi {<*string1*>**}{**<*string2*>**}**
The two substrings are first expanded. The condition is true if the two resulting strings are identical. For **eq** the comparison includes the case of letters, whereas for **eqi** the comparison is case-independent.

exists {<*file name*>**}**
The substring is first expanded and then interpreted as an absolute path. The condition is true if the named file (or directory) exists. The existence test is done by calling the *stat()* function. The use of the **exists** test in users' filter files may be locked out by the system administrator.

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.

forall{<*a list*>**}{**<*a condition*>**}**
forany{<*a list*>**}{**<*a condition*>**}**
These conditions iterate over a list. The first argument is expanded to form the list. By default, the list separator is a colon, but it can be changed by the normal method. The second argument is interpreted as a condition that is to be applied to each item in the list in turn. During the interpretation of the condition, the current list item is placed in a variable called *$item*.

- For **forany**, interpretation stops if the condition is true for any item, and the result of the whole condition is true. If the condition is false for all items in the list, the overall condition is false.
- For **forall**, interpretation stops if the condition is false for any item, and the result of the whole condition is false. If the condition is true for all items in the list, the overall condition is true.

Note that negation of **forany** means that the condition must be false for all items for the overall condition to succeed, and negation of **forall** means that the condition must be false for at least one item. In this example, the list separator is changed to a comma:

```
${if forany{<, $recipients}{match{$item}{^user3@}}{yes}{no}}
```

The value of *$item* is saved and restored while **forany** or **forall** is being processed, to enable these expansion items to be nested.

ge {<*string1*>**}{**<*string2*>**}**
gei {<*string1*>**}{**<*string2*>**}**
The two substrings are first expanded. The condition is true if the first string is lexically greater than or equal to the second string. For **ge** the comparison includes the case of letters, whereas for **gei** the comparison is case-independent.

gt {<*string1*>**}{**<*string2*>**}**
gti {<*string1*>**}{**<*string2*>**}**
The two substrings are first expanded. The condition is true if the first string is lexically greater than the second string. For **gt** the comparison includes the case of letters, whereas for **gti** the comparison is case-independent.

isip {<*string*>**}**
isip4 {<*string*>**}**
isip6 {<*string*>**}**
The substring is first expanded, and then tested to see if it has the form of an IP address. Both IPv4 and IPv6 addresses are valid for **isip**, whereas **isip4** and **isip6** test specifically for IPv4 or IPv6 addresses.

For an IPv4 address, the test is for four dot-separated components, each of which consists of from one to three digits. For an IPv6 address, up to eight colon-separated components are permitted, each containing from one to four hexadecimal digits. There may be fewer than eight components if an empty component (adjacent colons) is present. Only one empty component is permitted.

Note: The checks are just on the form of the address; actual numerical values are not considered. Thus, for example, 999.999.999.999 passes the IPv4 check. The main use of these tests is to distinguish between IP addresses and host names, or between IPv4 and IPv6 addresses. For example, you could use

```
${if isip4{$sender_host_address}...
```

to test which IP version an incoming SMTP connection is using.

ldapauth {<*ldap query*>**}**
This condition supports user authentication using LDAP. See section 9.13 for details of how to use LDAP in lookups and the syntax of queries. For this use, the query must contain a user name and password. The query itself is not used, and can be empty. The condition is true if the password is not empty, and the user name and password are accepted by the LDAP server. An empty password is rejected without calling LDAP because LDAP binds with an empty password are considered anonymous regardless of the username, and will succeed in most configurations. See chapter 33 for details of SMTP authentication, and chapter 34 for an example of how this can be used.

le {<*string1*>**}{**<*string2*>**}**
lei {<*string1*>**}{**<*string2*>**}**
The two substrings are first expanded. The condition is true if the first string is lexically less than or equal to the second string. For **le** the comparison includes the case of letters, whereas for **lei** the comparison is case-independent.

lt {<*string1*>**}{**<*string2*>**}**
lti {<*string1*>**}{**<*string2*>**}**
The two substrings are first expanded. The condition is true if the first string is lexically less than the second string. For **lt** the comparison includes the case of letters, whereas for **lti** the comparison is case-independent.

match {<*string1*>**}{**<*string2*>**}**
The two substrings are first expanded. The second is then treated as a regular expression and applied to the first. Because of the pre-expansion, if the regular expression contains dollar, or backslash characters, they must be escaped. 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*>. The easiest approach is to use the `\N` feature to disable expansion of the regular expression. For example,

```
${if match {$local_part}{\N^\d{3}\N} ...
```

If the whole expansion string is in double quotes, further escaping of backslashes is also required.

The condition is true if the regular expression match succeeds. The regular expression is not required to begin with a circumflex metacharacter, but if there is no circumflex, the expression is not anchored, and it may match anywhere in the subject, not just at the start. If you want the pattern to match at the end of the subject, you must include the `$` metacharacter at an appropriate point.

At the start of an **if** expansion the values of the numeric variable substitutions *$1* etc. are remembered. Obeying a **match** condition that succeeds causes them to be reset to the substrings of that condition and they will have these values during the expansion of the success string. At the end of

the **if** expansion, the previous values are restored. After testing a combination of conditions using **or**, the subsequent values of the numeric variables are those of the condition that succeeded.

match_address {<*string1*>**}{**<*string2*>**}**
See **match_local_part**.

match_domain {<*string1*>**}{**<*string2*>**}**
See **match_local_part**.

match_ip {<*string1*>**}{**<*string2*>**}**
This condition matches an IP address to a list of IP address patterns. It must be followed by two argument strings. The first (after expansion) must be an IP address or an empty string. The second (after expansion) is a restricted host list that can match only an IP address, not a host name. For example:

```
${if match_ip{$sender_host_address}{1.2.3.4:5.6.7.8}{...}{...}}
```

The specific types of host list item that are permitted in the list are:

- An IP address, optionally with a CIDR mask.
- A single asterisk, which matches any IP address.
- An empty item, which matches only if the IP address is empty. This could be useful for testing for a locally submitted message or one from specific hosts in a single test such as

  ```
  ${if match_ip{$sender_host_address}{:4.3.2.1:...}{...}{...}}
  ```

  where the first item in the list is the empty string.
- The item @[] matches any of the local host's interface addresses.
- Single-key lookups are assumed to be like "net-" style lookups in host lists, even if `net-` is not specified. There is never any attempt to turn the IP address into a host name. The most common type of linear search for **match_ip** is likely to be **iplsearch**, in which the file can contain CIDR masks. For example:

  ```
  ${if match_ip{$sender_host_address}{iplsearch;/some/file}...
  ```

  It is of course possible to use other kinds of lookup, and in such a case, you do need to specify the `net-` prefix if you want to specify a specific address mask, for example:

  ```
  ${if match_ip{$sender_host_address}{net24-dbm;/some/file}...
  ```

  However, unless you are combining a **match_ip** condition with others, it is just as easy to use the fact that a lookup is itself a condition, and write:

  ```
  ${lookup{${mask:$sender_host_address/24}}dbm{/a/file}...
  ```

Consult section 10.11 for further details of these patterns.

match_local_part {<*string1*>**}{**<*string2*>**}**
This condition, together with **match_address** and **match_domain**, make it possible to test domain, address, and local part lists within expansions. Each condition requires two arguments: an item and a list to match. A trivial example is:

```
${if match_domain{a.b.c}{x.y.z:a.b.c:p.q.r}{yes}{no}}
```

In each case, the second argument may contain any of the allowable items for a list of the appropriate type. Also, because the second argument (after expansion) is a standard form of list, it is possible to refer to a named list. Thus, you can use conditions like this:

```
${if match_domain{$domain}{+local_domains}{...
```

For address lists, the matching starts off caselessly, but the `+caseful` item can be used, as in all address lists, to cause subsequent items to have their local parts matched casefully. Domains are always matched caselessly.

Note: Host lists are *not* supported in this way. This is because hosts have two identities: a name and an IP address, and it is not clear how to specify cleanly how such a test would work. However, IP addresses can be matched using **match_ip**.

pam {<*string1*>**:**<*string2*>**:...}**
Pluggable Authentication Modules (**http://www.kernel.org/pub/linux/libs/pam/**) are a facility that is available in the latest releases of Solaris and in some GNU/Linux distributions. The Exim support, which is intended for use in conjunction with the SMTP AUTH command, is available only if Exim is compiled with

```
SUPPORT_PAM=yes
```

in *Local/Makefile*. You probably need to add **-lpam** to EXTRALIBS, and in some releases of GNU/Linux **-ldl** is also needed.

The argument string is first expanded, and the result must be a colon-separated list of strings. Leading and trailing white space is ignored. The PAM module is initialized with the service name "exim" and the user name taken from the first item in the colon-separated data string (<*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 consists of just two strings.

There can be problems if any of the strings are permitted to contain colon characters. In the usual way, these have to be doubled to avoid being taken as separators. If the data is being inserted from a variable, the **sg** expansion item can be used to double any existing colons. For example, the configuration of a LOGIN authenticator might contain this setting:

```
server_condition = ${if pam{$auth1:${sg{$auth2}{:}{::}}}}
```

For a PLAIN authenticator you could use:

```
server_condition = ${if pam{$auth2:${sg{$auth3}{:}{::}}}}
```

In some operating systems, PAM authentication can be done only from a process running as root. Since Exim is running as the Exim user when receiving messages, this means that PAM cannot be used directly in those systems. A patched version of the *pam_unix* module that comes with the Linux PAM package is available from **http://www.e-admin.de/pam_exim/**. The patched module allows one special uid/gid combination, in addition to root, to authenticate. If you build the patched module to allow the Exim user and group, PAM can then be used from an Exim authenticator.

pwcheck {<*string1*>**:**<*string2*>**}**
This condition supports user authentication using the Cyrus *pwcheck* daemon. This is one way of making it possible for passwords to be checked by a process that is not running as root. **Note**: The use of *pwcheck* is now deprecated. Its replacement is *saslauthd* (see below).

The pwcheck support is not included in Exim by default. You need to specify the location of the pwcheck daemon's socket in *Local/Makefile* before building Exim. For example:

```
CYRUS_PWCHECK_SOCKET=/var/pwcheck/pwcheck
```

You do not need to install the full Cyrus software suite in order to use the pwcheck daemon. You can compile and install just the daemon alone from the Cyrus SASL library. Ensure that *exim* is the only user that has access to the */var/pwcheck* directory.

The **pwcheck** condition takes one argument, which must be the user name and password, separated by a colon. For example, in a LOGIN authenticator configuration, you might have this:

```
server_condition = ${if pwcheck{$auth1:$auth2}}
```

queue_running
This condition, which has no data, is true during delivery attempts that are initiated by queue runner processes, and false otherwise.

radius {<*authentication string*>**}**
Radius authentication (RFC 2865) is supported in a similar way to PAM. You must set RADIUS_CONFIG_FILE in *Local/Makefile* to specify the location of the Radius client configuration file in order to build Exim with Radius support.

With just that one setting, Exim expects to be linked with the **radiusclient** library, using the original API. If you are using release 0.4.0 or later of this library, you need to set

```
RADIUS_LIB_TYPE=RADIUSCLIENTNEW
```

in *Local/Makefile* when building Exim. You can also link Exim with the **libradius** library that comes with FreeBSD. To do this, set

```
RADIUS_LIB_TYPE=RADLIB
```

in *Local/Makefile*, in addition to setting RADIUS_CONFIGURE_FILE. You may also have to supply a suitable setting in EXTRALIBS so that the Radius library can be found when Exim is linked.

The string specified by RADIUS_CONFIG_FILE is expanded and passed to the Radius client library, which calls the Radius server. The condition is true if the authentication is successful. For example:

```
server_condition = ${if radius{<arguments>}}
```

saslauthd {{<*user*>**}{**<*password*>**}{**<*service*>**}{**<*realm*>**}}**
This condition supports user authentication using the Cyrus *saslauthd* daemon. This replaces the older *pwcheck* daemon, which is now deprecated. Using this daemon is one way of making it possible for passwords to be checked by a process that is not running as root.

The saslauthd support is not included in Exim by default. You need to specify the location of the saslauthd daemon's socket in *Local/Makefile* before building Exim. For example:

```
CYRUS_SASLAUTHD_SOCKET=/var/state/saslauthd/mux
```

You do not need to install the full Cyrus software suite in order to use the saslauthd daemon. You can compile and install just the daemon alone from the Cyrus SASL library.

Up to four arguments can be supplied to the **saslauthd** condition, but only two are mandatory. For example:

```
server_condition = ${if saslauthd{{$auth1}{$auth2}}}
```

The service and the realm are optional (which is why the arguments are enclosed in their own set of braces). For details of the meaning of the service and realm, and how to run the daemon, consult the Cyrus documentation.

11.8 Combining expansion conditions

Several conditions can be tested at once by combining them using the **and** and **or** combination conditions. Note that **and** and **or** are complete conditions on their own, and precede their lists of sub-conditions. Each sub-condition must be enclosed in braces within the overall braces that contain the list. No repetition of **if** is used.

or {{<*cond1*>**}{**<*cond2*>**}...}**
The sub-conditions are evaluated from left to right. The condition is true if any one of the sub-conditions is true. For example,

```
${if or {{eq{$local_part}{spqr}}{eq{$domain}{testing.com}}}...
```

When a true sub-condition is found, the following ones are parsed but not evaluated. If there are several "match" sub-conditions the values of the numeric variables afterwards are taken from the first one that succeeds.

and {{<*cond1*>**}{**<*cond2*>**}...}**
The sub-conditions are evaluated from left to right. The condition is true if all of the sub-conditions are true. If there are several "match" sub-conditions, the values of the numeric variables

afterwards are taken from the last one. When a false sub-condition is found, the following ones are parsed but not evaluated.

11.9 Expansion variables

This section contains an alphabetical list of all the expansion variables. Some of them are available only when Exim is compiled with specific options such as support for TLS or the content scanning extension.

$0, *$1*, etc
: When a **match** 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. However, they do not retain their values afterwards; in fact, their previous values are restored at the end of processing an **if** item. The numerical variables may also be set externally by some other matching process which precedes the expansion of the string. For example, the commands available in Exim filter files include an **if** command with its own regular expression matching condition.

$acl_c...
: Values can be placed in these variables by the **set** modifier in an ACL. They can be given any name that starts with *$acl_c* and is at least six characters long, but the sixth character must be either a digit or an underscore. For example: *$acl_c5*, *$acl_c_mycount*. The values of the *$acl_c...* variables persist throughout the lifetime of an SMTP connection. They can be used to pass information between ACLs and between different invocations of the same ACL. When a message is received, the values of these variables are saved with the message, and can be accessed by filters, routers, and transports during subsequent delivery.

$acl_m...
: These variables are like the *$acl_c...* variables, except that their values are reset after a message has been received. Thus, if several messages are received in one SMTP connection, *$acl_m...* values are not passed on from one message to the next, as *$acl_c...* values are. The *$acl_m...* variables are also reset by MAIL, RSET, EHLO, HELO, and after starting a TLS session. When a message is received, the values of these variables are saved with the message, and can be accessed by filters, routers, and transports during subsequent delivery.

$acl_verify_message
: After an address verification has failed, this variable contains the failure message. It retains its value for use in subsequent modifiers. The message can be preserved by coding like this:

```
warn !verify = sender
     set acl_m0 = $acl_verify_message
```

You can use *$acl_verify_message* during the expansion of the **message** or **log_message** modifiers, to include information about the verification failure.

$address_data
: This variable is set by means of the **address_data** option in routers. The value then remains with the address while it is processed by subsequent routers and eventually a transport. If the transport is handling multiple addresses, the value from the first address is used. See chapter 15 for more details. **Note**: The contents of *$address_data* are visible in user filter files.

If *$address_data* is set when the routers are called from an ACL to verify a recipient address, the final value is still in the variable for subsequent conditions and modifiers of the ACL statement. If routing the address caused it to be redirected to just one address, the child address is also routed as part of the verification, and in this case the final value of *$address_data* is from the child's routing.

If *$address_data* is set when the routers are called from an ACL to verify a sender address, the final value is also preserved, but this time in *$sender_address_data*, to distinguish it from data from a recipient address.

In both cases (recipient and sender verification), the value does not persist after the end of the current ACL statement. If you want to preserve these values for longer, you can save them in ACL variables.

$address_file
When, as a result of aliasing, forwarding, or filtering, a message is directed to a specific file, this variable holds the name of the file when the transport is running. At other times, the variable is empty. 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 the text string `/home/r2d2/savemail`. For Sieve filters, the value may be "inbox" or a relative folder name. It is then up to the transport configuration to generate an appropriate absolute path to the relevant file.

$address_pipe
When, as a result of aliasing or forwarding, a message is directed to a pipe, this variable holds the pipe command when the transport is running.

$auth1 – $auth3
These variables are used in SMTP authenticators (see chapters 34–38). Elsewhere, they are empty.

$authenticated_id
When a server successfully authenticates a client it may be configured to preserve some of the authentication information in the variable *$authenticated_id* (see chapter 33). For example, a user/password authenticator configuration might preserve the user name for use in the routers. Note that this is not the same information that is saved in *$sender_host_authenticated*. When a message is submitted locally (that is, not over a TCP connection) the value of *$authenticated_id* is normally the login name of the calling process. However, a trusted user can override this by means of the **-oMai** command line option.

$authenticated_sender
When acting as a server, Exim takes note of the AUTH= parameter on an incoming SMTP MAIL command if it believes the sender is sufficiently trusted, as described in section 33.2. Unless the data is the string "<>", it is set as the authenticated sender of the message, and the value is available during delivery in the *$authenticated_sender* variable. If the sender is not trusted, Exim accepts the syntax of AUTH=, but ignores the data.

When a message is submitted locally (that is, not over a TCP connection), the value of *$authenticated_sender* is an address constructed from the login name of the calling process and *$qualify_domain*, except that a trusted user can override this by means of the **-oMas** command line option.

$authentication_failed
This variable is set to "1" in an Exim server if a client issues an AUTH command that does not succeed. Otherwise it is set to "0". This makes it possible to distinguish between "did not try to authenticate" (*$sender_host_authenticated* is empty and *$authentication_failed* is set to "0") and "tried to authenticate but failed" (*$sender_host_authenticated* is empty and *$authentication_failed* is set to "1"). Failure includes any negative response to an AUTH command, including (for example) an attempt to use an undefined mechanism.

$body_linecount
When a message is being received or delivered, this variable contains the number of lines in the message's body. See also *$message_linecount*.

$body_zerocount
When a message is being received or delivered, this variable contains the number of binary zero bytes in the message's body.

$bounce_recipient
This is set to the recipient address of a bounce message while Exim is creating it. It is useful if a customized bounce message text file is in use (see chapter 46).

$bounce_return_size_limit
This contains the value set in the **bounce_return_size_limit** option, rounded up to a multiple of 1000. It is useful when a customized error message text file is in use (see chapter 46).

$caller_gid
The real group id under which the process that called Exim was running. This is not the same as the group id of the originator of a message (see *$originator_gid*). If Exim re-execs itself, this variable in the new incarnation normally contains the Exim gid.

$caller_uid
The real user id under which the process that called Exim was running. This is not the same as the user id of the originator of a message (see *$originator_uid*). If Exim re-execs itself, this variable in the new incarnation normally contains the Exim uid.

$compile_date
The date on which the Exim binary was compiled.

$compile_number
The building process for Exim keeps a count of the number of times it has been compiled. This serves to distinguish different compilations of the same version of the program.

$demime_errorlevel
This variable is available when Exim is compiled with the content-scanning extension and the obsolete **demime** condition. For details, see section 41.6.

$demime_reason
This variable is available when Exim is compiled with the content-scanning extension and the obsolete **demime** condition. For details, see section 41.6.

$dnslist_domain
$dnslist_matched
$dnslist_text
$dnslist_value
When a DNS (black) list lookup succeeds, these variables are set to contain the following data from the lookup: the list's domain name, the key that was looked up, the contents of any associated TXT record, and the value from the main A record. See section 40.29 for more details.

$domain
When an address is being routed, or delivered on its own, this variable contains the domain. Uppercase letters in the domain are converted into lower case for *$domain*.

Global address rewriting happens when a message is received, so the value of *$domain* during routing and delivery is the value after rewriting. *$domain* is set during user filtering, but not during system filtering, because a message may have many recipients and the system filter is called just once.

When more than one address is being delivered at once (for example, several RCPT commands in one SMTP delivery), *$domain* is set only if they all have the same domain. Transports can be restricted to handling only one domain at a time if the value of *$domain* is required at transport time – this is the default for local transports. For further details of the environment in which local transports are run, see chapter 23.

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**.

The *$domain* variable is also used in some other circumstances:

- When an ACL is running for a RCPT command, *$domain* contains the domain of the recipient address. The domain of the *sender* address is in *$sender_address_domain* at both MAIL time and at RCPT time. *$domain* is not normally set during the running of the MAIL ACL. However, if the sender address is verified with a callout during the MAIL ACL, the sender domain is placed in *$domain* during the expansions of **hosts**, **interface**, and **port** in the *smtp* transport.
- When a rewrite item is being processed (see chapter 31), *$domain* contains the domain portion of the address that is being rewritten; it can be used in the expansion of the replacement address, for example, to rewrite domains by file lookup.
- With one important exception, whenever a domain list is being scanned, *$domain* contains the subject domain. **Exception**: When a domain list in a **sender_domains** condition in an ACL is

being processed, the subject domain is in *$sender_address_domain* and not in *$domain*. It works this way so that, in a RCPT ACL, the sender domain list can be dependent on the recipient domain (which is what is in *$domain* at this time).

- When the **smtp_etrn_command** option is being expanded, *$domain* contains the complete argument of the ETRN command (see section 45.8).

$domain_data
When the **domains** option on a router matches a domain by means of a lookup, the data read by the lookup is available during the running of the router as *$domain_data*. In addition, if the driver routes the address to a transport, the value is available in that transport. If the transport is handling multiple addresses, the value from the first address is used.

$domain_data is also set when the **domains** condition in an ACL matches a domain by means of a lookup. The data read by the lookup is available during the rest of the ACL statement. In all other situations, this variable expands to nothing.

$exim_gid
This variable contains the numerical value of the Exim group id.

$exim_path
This variable contains the path to the Exim binary.

$exim_uid
This variable contains the numerical value of the Exim user id.

$found_extension
This variable is available when Exim is compiled with the content-scanning extension and the obsolete **demime** condition. For details, see section 41.6.

$header_<name>
This is not strictly an expansion variable. It is expansion syntax for inserting the message header line with the given name. Note that the name must be terminated by colon or white space, because it may contain a wide variety of characters. Note also that braces must *not* be used.

$home
When the **check_local_user** option is set for a router, the user's home directory is placed in *$home* when the check succeeds. In particular, this means it is set during the running of users' filter files. A router may also explicitly set a home directory for use by a transport; this can be overridden by a setting on the transport itself.

When running a filter test via the **-bf** option, *$home* is set to the value of the environment variable HOME.

$host
If a router assigns an address to a transport (any transport), and passes a list of hosts with the address, the value of *$host* when the transport starts to run is the name of the first host on the list. Note that this applies both to local and remote transports.

For the *smtp* transport, if there is more than one host, the value of *$host* changes as the transport works its way through the list. In particular, when the *smtp* transport is expanding its options for encryption using TLS, or for specifying a transport filter (see chapter 24), *$host* contains the name of the host to which it is connected.

When used in the client part of an authenticator configuration (see chapter 33), *$host* contains the name of the server to which the client is connected.

$host_address
This variable is set to the remote host's IP address whenever *$host* is set for a remote connection. It is also set to the IP address that is being checked when the **ignore_target_hosts** option is being processed.

$host_data
If a **hosts** condition in an ACL is satisfied by means of a lookup, the result of the lookup is made available in the *$host_data* variable. This allows you, for example, to do things like this:

```
deny  hosts = net-lsearch;/some/file
message = $host_data
```

$host_lookup_deferred
This variable normally contains "0", as does *$host_lookup_failed*. When a message comes from a remote host and there is an attempt to look up the host's name from its IP address, and the attempt is not successful, one of these variables is set to "1".

- If the lookup receives a definite negative response (for example, a DNS lookup succeeded, but no records were found), *$host_lookup_failed* is set to "1".
- If there is any kind of problem during the lookup, such that Exim cannot tell whether or not the host name is defined (for example, a timeout for a DNS lookup), *$host_lookup_deferred* is set to "1".

Looking up a host's name from its IP address consists of more than just a single reverse lookup. Exim checks that a forward lookup of at least one of the names it receives from a reverse lookup yields the original IP address. If this is not the case, Exim does not accept the looked up name(s), and *$host_lookup_failed* is set to "1". Thus, being able to find a name from an IP address (for example, the existence of a PTR record in the DNS) is not sufficient on its own for the success of a host name lookup. If the reverse lookup succeeds, but there is a lookup problem such as a timeout when checking the result, the name is not accepted, and *$host_lookup_deferred* is set to "1". See also *$sender_host_name*.

$host_lookup_failed
See *$host_lookup_deferred*.

$inode
The only time this variable is set is while expanding the **directory_file** option in the *appendfile* transport. The variable contains the inode number of the temporary file which is about to be renamed. It can be used to construct a unique name for the file.

$interface_address
This is an obsolete name for *$received_ip_address*.

$interface_port
This is an obsolete name for *$received_port*.

$item
This variable is used during the expansion of **forall** and **forany** conditions (see section 11.7), and **filter**, **man**, and **reduce** items (see section 11.7). In other circumstances, it is empty.

$ldap_dn
This variable, which is available only when Exim is compiled with LDAP support, contains the DN from the last entry in the most recently successful LDAP lookup.

$load_average
This variable contains the system load average, multiplied by 1000 so that it is an integer. For example, if the load average is 0.21, the value of the variable is 210. The value is recomputed every time the variable is referenced.

$local_part
When an address is being routed, or delivered on its own, this variable contains the local part. When a number of addresses are being delivered together (for example, multiple RCPT commands in an SMTP session), *$local_part* is not set.

Global address rewriting happens when a message is received, so the value of *$local_part* during routing and delivery is the value after rewriting. *$local_part* is set during user filtering, but not during system filtering, because a message may have many recipients and the system filter is called just once.

If a local part prefix or suffix has been recognized, it is not included in the value of *$local_part* during routing and subsequent delivery. The values of any prefix or suffix are in *$local_part_prefix* and *$local_part_suffix*, respectively.

When a message is being delivered to a file, pipe, or autoreply transport as a result of aliasing or forwarding, *$local_part* is set to the local part of the parent address, not to the file name or command (see *$address_file* and *$address_pipe*).

When an ACL is running for a RCPT command, *$local_part* contains the local part of the recipient address.

When a rewrite item is being processed (see chapter 31), *$local_part* contains the local part of the address that is being rewritten; it can be used in the expansion of the replacement address, for example.

In all cases, all quoting is removed from the local part. For example, for both the addresses

```
"abc:xyz"@test.example
abc\:xyz@test.example
```

the value of *$local_part* is

```
abc:xyz
```

If you use *$local_part* to create another address, you should always wrap it inside a quoting operator. For example, in a *redirect* router you could have:

```
data = ${quote_local_part:$local_part}@new.domain.example
```

Note: The value of *$local_part* is normally lower cased. If you want to process local parts in a case-dependent manner in a router, you can set the **caseful_local_part** option (see chapter 15).

$local_part_data
When the **local_parts** option on a router matches a local part by means of a lookup, the data read by the lookup is available during the running of the router as *$local_part_data*. In addition, if the driver routes the address to a transport, the value is available in that transport. If the transport is handling multiple addresses, the value from the first address is used.

$local_part_data is also set when the **local_parts** condition in an ACL matches a local part by means of a lookup. The data read by the lookup is available during the rest of the ACL statement. In all other situations, this variable expands to nothing.

$local_part_prefix
When an address is being routed or delivered, and a specific prefix for the local part was recognized, it is available in this variable, having been removed from *$local_part*.

$local_part_suffix
When an address is being routed or delivered, and a specific suffix for the local part was recognized, it is available in this variable, having been removed from *$local_part*.

$local_scan_data
This variable contains the text returned by the *local_scan()* function when a message is received. See chapter 42 for more details.

$local_user_gid
See *$local_user_uid*.

$local_user_uid
This variable and *$local_user_gid* are set to the uid and gid after the **check_local_user** router precondition succeeds. This means that their values are available for the remaining preconditions (**senders**, **require_files**, and **condition**), for the **address_data** expansion, and for any router-specific expansions. At all other times, the values in these variables are `(uid_t)(-1)` and `(gid_t)(-1)`, respectively.

$localhost_number
This contains the expanded value of the **localhost_number** option. The expansion happens after the main options have been read.

$log_inodes
The number of free inodes in the disk partition where Exim's log files are being written. The value is recalculated whenever the variable is referenced. If the relevant file system does not have the concept of inodes, the value of is -1. See also the **check_log_inodes** option.

$log_space
The amount of free space (as a number of kilobytes) in the disk partition where Exim's log files are being written. The value is recalculated whenever the variable is referenced. If the operating system does not have the ability to find the amount of free space (only true for experimental systems), the space value is -1. See also the **check_log_space** option.

$mailstore_basename
This variable is set only when doing deliveries in "mailstore" format in the *appendfile* transport. During the expansion of the **mailstore_prefix**, **mailstore_suffix**, **message_prefix**, and **message_suffix** options, it contains the basename of the files that are being written, that is, the name without the ".tmp", ".env", or ".msg" suffix. At all other times, this variable is empty.

$malware_name
This variable is available when Exim is compiled with the content-scanning extension. It is set to the name of the virus that was found when the ACL **malware** condition is true (see section 41.1).

$max_received_linelength
This variable contains the number of bytes in the longest line that was received as part of the message, not counting the line termination character(s).

$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 put into the variable is set by the **message_body_visible** configuration option; the default is 500.

By default, newlines are converted into spaces in *$message_body*, to make it easier to search for phrases that might be split over a line break. However, this can be disabled by setting **message_body_newlines** to be true. Binary zeros are always converted into spaces.

$message_body_end
This variable contains the final portion of a message's body while it is being delivered. The format and maximum size are as for *$message_body*.

$message_body_size
When a message is being delivered, this variable contains the size of the body in bytes. The count starts from the character after the blank line that separates the body from the header. Newlines are included in the count. See also *$message_size*, *$body_linecount*, and *$body_zerocount*.

$message_exim_id
When a message is being received or delivered, this variable contains the unique message id that is generated and used by Exim to identify the message. An id is not created for a message until after its header has been successfully received. **Note**: This is *not* the contents of the *Message-ID:* header line; it is the local id that Exim assigns to the message, for example: `1BXTIK-0001yO-VA`.

$message_headers
This variable contains a concatenation of all the header lines when a message is being processed, except for lines added by routers or transports. The header lines are separated by newline characters. Their contents are decoded in the same way as a header line that is inserted by **bheader**.

$message_headers_raw
This variable is like *$message_headers* except that no processing of the contents of header lines is done.

$message_id
This is an old name for *$message_exim_id*, which is now deprecated.

$message_linecount
This variable contains the total number of lines in the header and body of the message. Compare *$body_linecount*, which is the count for the body only. During the DATA and content-scanning ACLs, *$message_linecount* contains the number of lines received. Before delivery happens (that is, before filters, routers, and transports run) the count is increased to include the *Received:* header line that Exim standardly adds, and also any other header lines that are added by ACLs. The blank line that separates the message header from the body is not counted. Here is an example of the use of this variable in a DATA ACL:

```
deny message   = Too many lines in message header
     condition = \
      ${if <{250}{${eval:$message_linecount - $body_linecount}}}
```

In the MAIL and RCPT ACLs, the value is zero because at that stage the message has not yet been received.

$message_size
When a message is being processed, this variable contains its size in bytes. In most cases, the size includes those headers that were received with the message, but not those (such as *Envelope-to:*) that are added to individual deliveries as they are written. However, there is one special case: during the expansion of the **maildir_tag** option in the *appendfile* transport while doing a delivery in maildir format, the value of *$message_size* is the precise size of the file that has been written. See also *$message_body_size*, *$body_linecount*, and *$body_zerocount*.

While running an ACL at the time of an SMTP RCPT command, *$message_size* contains the size supplied on the MAIL command, or -1 if no size was given. The value may not, of course, be truthful.

$mime_xxx
A number of variables whose names start with *$mime* are available when Exim is compiled with the content-scanning extension. For details, see section 41.4.

$n0 – $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 (lower cased). This differs from *$parent_domain* only when there is more than one level of aliasing or forwarding. When more than one address is being delivered in a single transport run, *$original_domain* is not set.

If a new address is created by means of a **deliver** command in a system filter, it is set up with an artificial "parent" address. This has the local part *system-filter* and the default qualify domain.

$original_local_part
When a top-level address is being processed for delivery, this contains the same value as *$local_part*, unless a prefix or suffix was removed from the local part, because *$original_local_part* always contains the full local part. When a "child" address (for example, generated by an alias, forward, or filter file) is being processed, this variable contains the full local part of the original address.

If the router that did the redirection processed the local part case-insensitively, the value in *$original_local_part* is in lower case. This variable differs from *$parent_local_part* only when there is more than one level of aliasing or forwarding. When more than one address is being delivered in a single transport run, *$original_local_part* is not set.

If a new address is created by means of a **deliver** command in a system filter, it is set up with an artificial "parent" address. This has the local part *system-filter* and the default qualify domain.

$originator_gid
This variable contains the value of *$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.

$parent_domain
This variable is similar to *$original_domain* (see above), except that it refers to the immediately preceding parent address.

$parent_local_part
This variable is similar to *$original_local_part* (see above), except that it refers to the immediately preceding parent address.

$pid
This variable contains the current process id.

$pipe_addresses
This is not an expansion variable, but is mentioned here because the string `$pipe_addresses` is handled specially in the command specification for the *pipe* transport (chapter 29) and in transport filters (described under **transport_filter** in chapter 24). It cannot be used in general expansion strings, and provokes an "unknown variable" error if encountered.

$primary_hostname
This variable contains the value set by **primary_hostname** in the configuration file, or read by the *uname()* function. If *uname()* returns a single-component name, Exim calls *gethostbyname()* (or *getipnodebyname()* where available) in an attempt to acquire a fully qualified host name. See also *$smtp_active_hostname*.

$prvscheck_address
This variable is used in conjunction with the **prvscheck** expansion item, which is described in sections 11.5 and 40.47.

$prvscheck_keynum
This variable is used in conjunction with the **prvscheck** expansion item, which is described in sections 11.5 and 40.47.

$prvscheck_result
This variable is used in conjunction with the **prvscheck** expansion item, which is described in sections 11.5 and 40.47.

$qualify_domain
The value set for the **qualify_domain** option in the configuration file.

$qualify_recipient
The value set for the **qualify_recipient** option in the configuration file, or if not set, the value of *$qualify_domain*.

$rcpt_count
When a message is being received by SMTP, this variable contains the number of RCPT commands received for the current message. If this variable is used in a RCPT ACL, its value includes the current command.

$rcpt_defer_count
When a message is being received by SMTP, this variable contains the number of RCPT commands in the current message that have previously been rejected with a temporary (4*xx*) response.

$rcpt_fail_count
When a message is being received by SMTP, this variable contains the number of RCPT commands in the current message that have previously been rejected with a permanent (5*xx*) response.

$received_count
This variable contains the number of *Received:* header lines in the message, including the one added by Exim (so its value is always greater than zero). It is available in the DATA ACL, the non-SMTP ACL, and while routing and delivering.

$received_for
If there is only a single recipient address in an incoming message, this variable contains that address when the *Received:* header line is being built. The value is copied after recipient rewriting has happened, but before the *local_scan()* function is run.

$received_ip_address
As soon as an Exim server starts processing an incoming TCP/IP connection, this variable is set to the address of the local IP interface, and *$received_port* is set to the local port number. (The remote IP address and port are in *$sender_host_address* and *$sender_host_port*.) When testing with **-bh**, the port value is -1 unless it has been set using the **-oMi** command line option.

As well as being useful in ACLs (including the "connect" ACL), these variable could be used, for example, to make the file name for a TLS certificate depend on which interface and/or port is being used for the incoming connection. The values of *$received_ip_address* and *$received_port* are saved with any messages that are received, thus making these variables available at delivery time.

Note: There are no equivalent variables for outgoing connections, because the values are unknown (unless they are explicitly set by options of the *smtp* transport).

$received_port
See *$received_ip_address*.

$received_protocol
When a message is being processed, this variable contains the name of the protocol by which it was received. Most of the names used by Exim are defined by RFCs 821, 2821, and 3848. They start with "smtp" (the client used HELO) or "esmtp" (the client used EHLO). This can be followed by "s" for secure (encrypted) and/or "a" for authenticated. Thus, for example, if the protocol is set to "esmtpsa", the message was received over an encrypted SMTP connection and the client was successfully authenticated.

Exim uses the protocol name "smtps" for the case when encryption is automatically set up on connection without the use of STARTTLS (see **tls_on_connect_ports**), and the client uses HELO to initiate the encrypted SMTP session. The name "smtps" is also used for the rare situation where the client initially uses EHLO, sets up an encrypted connection using STARTTLS, and then uses HELO afterwards.

The **-oMr** option provides a way of specifying a custom protocol name for messages that are injected locally by trusted callers. This is commonly used to identify messages that are being re-injected after some kind of scanning.

$received_time
This variable contains the date and time when the current message was received, as a number of seconds since the start of the Unix epoch.

$recipient_data
This variable is set after an indexing lookup success in an ACL **recipients** condition. It contains the data from the lookup, and the value remains set until the next **recipients** test. Thus, you can do things like this:

```
require recipients  = cdb*@;/some/file
deny    some further test involving $recipient_data
```

Warning: This variable is set only when a lookup is used as an indexing method in the address list, using the semicolon syntax as in the example above. The variable is not set for a lookup that is used as part of the string expansion that all such lists undergo before being interpreted.

$recipient_verify_failure
In an ACL, when a recipient verification fails, this variable contains information about the failure. It is set to one of the following words:

- "qualify": The address was unqualified (no domain), and the message was neither local nor came from an exempted host.
- "route": Routing failed.

- "mail": Routing succeeded, and a callout was attempted; rejection occurred at or before the MAIL command (that is, on initial connection, HELO, or MAIL).
- "recipient": The RCPT command in a callout was rejected.
- "postmaster": The postmaster check in a callout was rejected.

The main use of this variable is expected to be to distinguish between rejections of MAIL and rejections of RCPT.

$recipients
This variable contains a list of envelope recipients for a message. A comma and a space separate the addresses in the replacement text. However, the variable is not generally available, to prevent exposure of Bcc recipients in unprivileged users' filter files. You can use *$recipients* only in these cases:

(1) In a system filter file.

(2) In the ACLs associated with the DATA command and with non-SMTP messages, that is, the ACLs defined by **acl_smtp_predata**, **acl_smtp_data**, **acl_smtp_mime**, **acl_not_smtp_start**, **acl_not_smtp**, and **acl_not_smtp_mime**.

(3) From within a *local_scan()* function.

$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. While a message is being received over SMTP, the number increases for each accepted recipient. It can be referenced in an ACL.

$regex_match_string
This variable is set to contain the matching regular expression after a **regex** ACL condition has matched (see section 41.5).

$reply_address
When a message is being processed, this variable contains the contents of the *Reply-To:* header line if one exists and it is not empty, or otherwise the contents of the *From:* header line. Apart from the removal of leading white space, the value is not processed in any way. In particular, no RFC 2047 decoding or character code translation takes place.

$return_path
When a message is being delivered, this variable contains the return path – the sender field that will be sent as part of the envelope. It is not enclosed in <> characters. At the start of routing an address, *$return_path* has the same value as *$sender_address*, but if, for example, an incoming message to a mailing list has been expanded by a router which specifies a different address for bounce messages, *$return_path* subsequently contains the new bounce address, whereas *$sender_address* always contains the original sender address that was received with the message. In other words, *$sender_address* contains the incoming envelope sender, and *$return_path* contains the outgoing envelope sender.

$return_size_limit
This is an obsolete name for *$bounce_return_size_limit*.

$runrc
This variable contains the return code from a command that is run by the **${run...}** expansion item. **Warning**: In a router or transport, you cannot assume the order in which option values are expanded, except for those preconditions whose order of testing is documented. Therefore, you cannot reliably expect to set *$runrc* by the expansion of one option, and use it in another.

$self_hostname
When an address is routed to a supposedly remote host that turns out to be the local host, what happens is controlled by the **self** generic router option. One of its values causes the address to be passed to another router. When this happens, *$self_hostname* is set to the name of the local host that the original router encountered. In other circumstances its contents are null.

$sender_address
When a message is being processed, this variable contains the sender's address that was received in the message's envelope. The case of letters in the address is retained, in both the local part and the domain. For bounce messages, the value of this variable is the empty string. See also *$return_path*.

$sender_address_data
If *$address_data* is set when the routers are called from an ACL to verify a sender address, the final value is preserved in *$sender_address_data*, to distinguish it from data from a recipient address. The value does not persist after the end of the current ACL statement. If you want to preserve it for longer, you can save it in an ACL variable.

$sender_address_domain
The domain portion of *$sender_address*.

$sender_address_local_part
The local part portion of *$sender_address*.

$sender_data
This variable is set after a lookup success in an ACL **senders** condition or in a router **senders** option. It contains the data from the lookup, and the value remains set until the next **senders** test. Thus, you can do things like this:

```
require senders      = cdb*@;/some/file
deny    some further test involving $sender_data
```

Warning: This variable is set only when a lookup is used as an indexing method in the address list, using the semicolon syntax as in the example above. The variable is not set for a lookup that is used as part of the string expansion that all such lists undergo before being interpreted.

$sender_fullhost
When a message is received from a remote host, this variable contains the host name and IP address in a single string. It ends with the IP address in square brackets, followed by a colon and a port number if the logging of ports is enabled. The format of the rest of the string depends on whether the host issued a HELO or EHLO SMTP command, and whether the host name 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 host name at the start of the string is a verified host name; if this is not present, verification either failed or was not requested. A host name in parentheses is the argument of a HELO or EHLO command. This is omitted if it is identical to the verified host name or to the host's IP address in square brackets.

$sender_helo_name
When a message is received from a remote host that has issued a HELO or EHLO command, 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 is received from a remote host, this variable contains that host's IP address. For locally submitted messages, it is empty.

$sender_host_authenticated
This variable contains the name (not the public name) of the authenticator driver that successfully authenticated the client from which the message was received. It is empty if there was no successful authentication. See also *$authenticated_id*.

$sender_host_name
When a message is received from a remote host, this variable contains the host's name as obtained by looking up its IP address. For messages received by other means, this variable is empty.

If the host name has not previously been looked up, a reference to *$sender_host_name* triggers a lookup (for messages from remote hosts). A looked up name is accepted only if it leads back to the original IP address via a forward lookup. If either the reverse or the forward lookup fails to find any data, or if the forward lookup does not yield the original IP address, *$sender_host_name* remains empty, and *$host_lookup_failed* is set to "1".

However, if either of the lookups cannot be completed (for example, there is a DNS timeout), *$host_lookup_deferred* is set to "1", and *$host_lookup_failed* remains set to "0".

Once *$host_lookup_failed* is set to "1", Exim does not try to look up the host name again if there is a subsequent reference to *$sender_host_name* in the same Exim process, but it does try again if *$host_lookup_deferred* is set to "1".

Exim does not automatically look up every calling host's name. If you want maximum efficiency, you should arrange your configuration so that it avoids these lookups altogether. The lookup happens only if one or more of the following are true:

- A string containing *$sender_host_name* is expanded.
- The calling host matches the list in **host_lookup**. In the default configuration, this option is set to *, so it must be changed if lookups are to be avoided. (In the code, the default for **host_lookup** is unset.)
- Exim needs the host name in order to test an item in a host list. The items that require this are described in sections 10.13 and 10.16.
- The calling host matches **helo_try_verify_hosts** or **helo_verify_hosts**. In this case, the host name is required to compare with the name quoted in any EHLO or HELO commands that the client issues.
- The remote host issues a EHLO or HELO command that quotes one of the domains in **helo_lookup_domains**. The default value of this option is

  ```
  helo_lookup_domains = @ : @[]
  ```

  which causes a lookup if a remote host (incorrectly) gives the server's name or IP address in an EHLO or HELO command.

$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 is 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.

$sender_rate_xxx
A number of variables whose names begin *$sender_rate_* are set as part of the **ratelimit** ACL condition. Details are given in section 40.35.

$sender_rcvhost
This is provided specifically for use in *Received:* headers. It starts with either the verified host name (as obtained from a reverse DNS lookup) or, if there is no verified host name, the IP address in square brackets. After that there may be text in parentheses. When the first item is a verified host name, the first thing in the parentheses is the IP address in square brackets, followed by a colon and a port number if port logging is enabled. When the first item is an IP address, the port is recorded as "port=*xxxx*" inside the parentheses.

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 host name 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.

$sender_verify_failure
In an ACL, when a sender verification fails, this variable contains information about the failure. The details are the same as for *$recipient_verify_failure*.

$sending_ip_address
This variable is set whenever an outgoing SMTP connection to another host has been set up. It contains the IP address of the local interface that is being used. This is useful if a host that has

more than one IP address wants to take on different personalities depending on which one is being used. For incoming connections, see *$received_ip_address*.

$sending_port
This variable is set whenever an outgoing SMTP connection to another host has been set up. It contains the local port that is being used. For incoming connections, see *$received_port*.

$smtp_active_hostname
During an incoming SMTP session, this variable contains the value of the active host name, as specified by the **smtp_active_hostname** option. The value of *$smtp_active_hostname* is saved with any message that is received, so its value can be consulted during routing and delivery.

$smtp_command
During the processing of an incoming SMTP command, this variable contains the entire command. This makes it possible to distinguish between HELO and EHLO in the HELO ACL, and also to distinguish between commands such as these:

```
MAIL FROM:<>
MAIL FROM: <>
```

For a MAIL command, extra parameters such as SIZE can be inspected. For a RCPT command, the address in *$smtp_command* is the original address before any rewriting, whereas the values in *$local_part* and *$domain* are taken from the address after SMTP-time rewriting.

$smtp_command_argument
While an ACL is running to check an SMTP command, this variable contains the argument, that is, the text that follows the command name, with leading white space removed. Following the introduction of *$smtp_command*, this variable is somewhat redundant, but is retained for backwards compatibility.

$smtp_count_at_connection_start
This variable is set greater than zero only in processes spawned by the Exim daemon for handling incoming SMTP connections. The name is deliberately long, in order to emphasize what the contents are. When the daemon accepts a new connection, it increments this variable. A copy of the variable is passed to the child process that handles the connection, but its value is fixed, and never changes. It is only an approximation of how many incoming connections there actually are, because many other connections may come and go while a single connection is being processed. When a child process terminates, the daemon decrements its copy of the variable.

$sn0 – $sn9
These variables are copies of the values of the *$n0 – $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.

$spam_xxx
A number of variables whose names start with *$spam* are available when Exim is compiled with the content-scanning extension. For details, see section 41.2.

$spool_directory
The name of Exim's spool directory.

$spool_inodes
The number of free inodes in the disk partition where Exim's spool files are being written. The value is recalculated whenever the variable is referenced. If the relevant file system does not have the concept of inodes, the value of is -1. See also the **check_spool_inodes** option.

$spool_space
The amount of free space (as a number of kilobytes) in the disk partition where Exim's spool files are being written. The value is recalculated whenever the variable is referenced. If the operating system does not have the ability to find the amount of free space (only true for experimental systems), the space value is -1. For example, to check in an ACL that there is at least 50 megabytes free on the spool, you could write:

```
condition = ${if > {$spool_space}{50000}}
```

See also the **check_spool_space** option.

$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, which can be found in the separate document entitled *Exim's interfaces to mail filtering*.

$tls_certificate_verified
This variable is set to "1" if a TLS certificate was verified when the message was received, and "0" otherwise.

$tls_cipher
When a message is received from a remote host over an encrypted SMTP connection, this variable is set to the cipher suite that was negotiated, for example DES-CBC3-SHA. In other circumstances, in particular, for message received over unencrypted connections, the variable is empty. Testing *$tls_cipher* for emptiness is one way of distinguishing between encrypted and non-encrypted connections during ACL processing.

The *$tls_cipher* variable retains its value during message delivery, except when an outward SMTP delivery takes place via the *smtp* transport. In this case, *$tls_cipher* is cleared before any outgoing SMTP connection is made, and then set to the outgoing cipher suite if one is negotiated. See chapter 39 for details of TLS support and chapter 30 for details of the *smtp* transport.

$tls_peerdn
When a message is received from a remote host over an encrypted SMTP connection, and Exim is configured to request a certificate from the client, the value of the Distinguished Name of the certificate is made available in the *$tls_peerdn* during subsequent processing. Like *$tls_cipher*, the value is retained during message delivery, except during outbound SMTP deliveries.

$tod_bsdinbox
The time of day and the date, in the format required for BSD-style mailbox files, for example: Thu Oct 17 17:14:09 1995.

$tod_epoch
The time and date as a number of seconds since the start of the Unix epoch.

$tod_full
A full version of the time and date, for example: Wed, 16 Oct 1995 09:51:40 +0100. The timezone is always given as a numerical offset from UTC, with positive values used for timezones that are ahead (east) of UTC, and negative values for those that are behind (west).

$tod_log
The time and date in the format used for writing Exim's log files, for example: 1995-10-12 15:32:29, but without a timezone.

$tod_logfile
This variable contains the date in the format yyyymmdd. This is the format that is used for datestamping log files when **log_file_path** contains the `%D` flag.

$tod_zone
This variable contains the numerical value of the local timezone, for example: -0500.

$tod_zulu
This variable contains the UTC date and time in "Zulu" format, as specified by ISO 8601, for example: 20030221154023Z.

$value
This variable contains the result of an expansion lookup, extraction operation, or external command, as described above. It is also used during a **reduce** expansion.

$version_number
The version number of Exim.

$warn_message_delay
This variable is set only during the creation of a message warning about a delivery delay. Details of its use are explained in section 46.2.

$warn_message_recipients
This variable is set only during the creation of a message warning about a delivery delay. Details of its use are explained in section 46.2.

12. Embedded Perl

Exim can be built to include an embedded Perl interpreter. When this is done, Perl subroutines can be called as part of the string expansion process. To make use of the Perl support, you need version 5.004 or later of Perl installed on your system. To include the embedded interpreter in the Exim binary, include the line

```
EXIM_PERL = perl.o
```

in your *Local/Makefile* and then build Exim in the normal way.

12.1 Setting up so Perl can be used

Access to Perl subroutines is via a global configuration option called **perl_startup** and an expansion string operator **${perl ...}**. If there is no **perl_startup** option in the Exim configuration file then no Perl interpreter is started and there is almost no overhead for Exim (since none of the Perl library will be paged in unless used). If there is a **perl_startup** option then the associated value is taken to be Perl code which is executed in a newly created Perl interpreter.

The value of **perl_startup** 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 like

```
perl_startup = do '/etc/exim.pl'
```

where */etc/exim.pl* is Perl code which defines any subroutines you want to use from Exim. Exim can be configured either to start up a Perl interpreter as soon as it is entered, or to wait until the first time it is needed. Starting the interpreter at the beginning ensures that it is done while Exim still has its setuid privilege, but can impose an unnecessary overhead if Perl is not in fact used in a particular run. Also, note that this does not mean that Exim is necessarily running as root when Perl is called at a later time. By default, the interpreter is started only when it is needed, but this can be changed in two ways:

- Setting **perl_at_start** (a boolean option) in the configuration requests a startup when Exim is entered.
- The command line option **-ps** also requests a startup when Exim is entered, overriding the setting of **perl_at_start**.

There is also a command line option **-pd** (for delay) which suppresses the initial startup, even if **perl_at_start** is set.

12.2 Calling Perl subroutines

When the configuration file includes a **perl_startup** option you can make use of the string expansion item to call the Perl subroutines that are defined by the **perl_startup** code. The operator is used in any of the following forms:

```
${perl{foo}}
${perl{foo}{argument}}
${perl{foo}{argument1}{argument2} ... }
```

which calls the subroutine **foo** with the given arguments. A maximum of eight arguments may be passed. Passing more than this results in an expansion failure with an error message of the form

```
Too many arguments passed to Perl subroutine "foo" (max is 8)
```

The return value of the Perl subroutine is evaluated in a scalar context before it is passed back to Exim to be inserted into the expanded string. If the return value is *undef*, the expansion is forced to fail 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**.

12.3 Calling Exim functions from Perl

Within any Perl code called from Exim, the function *Exim::expand_string()* is available to call back into Exim's string expansion function. For example, the Perl code

```
my $lp = Exim::expand_string('$local_part');
```

makes the current Exim *$local_part* available in the Perl variable *$lp*. Note those are single quotes and not double quotes to protect against *$local_part* being interpolated as a Perl variable.

If the string expansion is forced to fail by a "fail" item, the result of *Exim::expand_string()* is **undef**. If there is a syntax error in the expansion string, the Perl call from the original expansion string fails with an appropriate error message, in the same way as if **die** were used.

Two other Exim functions are available for use from within Perl code. *Exim::debug_write()* writes a string to the standard error stream if Exim's debugging is enabled. If you want a newline at the end, you must supply it. *Exim::log_write()* writes a string to Exim's main log, adding a leading timestamp. In this case, you should not supply a terminating newline.

12.4 Use of standard output and error by Perl

You should not write to the standard error or output streams from within your Perl code, as it is not defined how these are set up. In versions of Exim before 4.50, it is possible for the standard output or error to refer to the SMTP connection during message reception via the daemon. Writing to this stream is certain to cause chaos. From Exim 4.50 onwards, the standard output and error streams are connected to */dev/null* in the daemon. The chaos is avoided, but the output is lost.

The Perl **warn** statement writes to the standard error stream by default. Calls to **warn** may be embedded in Perl modules that you use, but over which you have no control. When Exim starts up the Perl interpreter, it arranges for output from the **warn** statement to be written to the Exim main log. You can change this by including appropriate Perl magic somewhere in your Perl code. For example, to discard **warn** output completely, you need this:

```
$SIG{__WARN__} = sub { };
```

Whenever a **warn** is obeyed, the anonymous subroutine is called. In this example, the code for the subroutine is empty, so it does nothing, but you can include any Perl code that you like. The text of the **warn** message is passed as the first subroutine argument.

13. Starting the daemon and the use of network interfaces

A host that is connected to a TCP/IP network may have one or more physical hardware network interfaces. Each of these interfaces may be configured as one or more "logical" interfaces, which are the entities that a program actually works with. Each of these logical interfaces is associated with an IP address. In addition, TCP/IP software supports "loopback" interfaces (127.0.0.1 in IPv4 and ::1 in IPv6), which do not use any physical hardware. Exim requires knowledge about the host's interfaces for use in three different circumstances:

(1) When a listening daemon is started, Exim needs to know which interfaces and ports to listen on.

(2) When Exim is routing an address, it needs to know which IP addresses are associated with local interfaces. This is required for the correct processing of MX lists by removing the local host and others with the same or higher priority values. Also, Exim needs to detect cases when an address is routed to an IP address that in fact belongs to the local host. Unless the **self** router option or the **allow_localhost** option of the smtp transport is set (as appropriate), this is treated as an error situation.

(3) When Exim connects to a remote host, it may need to know which interface to use for the outgoing connection.

Exim's default behaviour is likely to be appropriate in the vast majority of cases. If your host has only one interface, and you want all its IP addresses to be treated in the same way, and you are using only the standard SMTP port, you should not need to take any special action. The rest of this chapter does not apply to you.

In a more complicated situation you may want to listen only on certain interfaces, or on different ports, and for this reason there are a number of options that can be used to influence Exim's behaviour. The rest of this chapter describes how they operate.

When a message is received over TCP/IP, the interface and port that were actually used are set in *$received_ip_address* and *$received_port*.

13.1 Starting a listening daemon

When a listening daemon is started (by means of the **-bd** command line option), the interfaces and ports on which it listens are controlled by the following options:

- **daemon_smtp_ports** contains a list of default ports. (For backward compatibility, this option can also be specified in the singular.)
- **local_interfaces** contains list of interface IP addresses on which to listen. Each item may optionally also specify a port.

The default list separator in both cases is a colon, but this can be changed as described in section 6.19. When IPv6 addresses are involved, it is usually best to change the separator to avoid having to double all the colons. For example:

```
local_interfaces = <; 127.0.0.1 ; \
                      192.168.23.65 ; \
                      ::1 ; \
                      3ffe:ffff:836f::fe86:a061
```

There are two different formats for specifying a port along with an IP address in **local_interfaces**:

(1) The port is added onto the address with a dot separator. For example, to listen on port 1234 on two different IP addresses:

```
local_interfaces = <; 192.168.23.65.1234 ; \
                      3ffe:ffff:836f::fe86:a061.1234
```

(2) The IP address is enclosed in square brackets, and the port is added with a colon separator, for example:

```
local_interfaces = <; [192.168.23.65]:1234 ; \
                      [3ffe:ffff:836f::fe86:a061]:1234
```

When a port is not specified, the value of **daemon_smtp_ports** is used. The default setting contains just one port:

```
daemon_smtp_ports = smtp
```

If more than one port is listed, each interface that does not have its own port specified listens on all of them. Ports that are listed in **daemon_smtp_ports** can be identified either by name (defined in */etc/services*) or by number. However, when ports are given with individual IP addresses in **local_interfaces**, only numbers (not names) can be used.

13.2 Special IP listening addresses

The addresses 0.0.0.0 and ::0 are treated specially. They are interpreted as "all IPv4 interfaces" and "all IPv6 interfaces", respectively. In each case, Exim tells the TCP/IP stack to "listen on all IPv*x* interfaces" instead of setting up separate listening sockets for each interface. The default value of **local_interfaces** is

```
local_interfaces = 0.0.0.0
```

when Exim is built without IPv6 support; otherwise it is:

```
local_interfaces = <; ::0 ; 0.0.0.0
```

Thus, by default, Exim listens on all available interfaces, on the SMTP port.

13.3 Overriding local_interfaces and daemon_smtp_ports

The **-oX** command line option can be used to override the values of **daemon_smtp_ports** and/or **local_interfaces** for a particular daemon instance. Another way of doing this would be to use macros and the **-D** option. However, **-oX** can be used by any admin user, whereas modification of the runtime configuration by **-D** is allowed only when the caller is root or exim.

The value of **-oX** is a list of items. The default colon separator can be changed in the usual way if required. If there are any items that do not contain dots or colons (that is, are not IP addresses), the value of **daemon_smtp_ports** is replaced by the list of those items. If there are any items that do contain dots or colons, the value of **local_interfaces** is replaced by those items. Thus, for example,

```
-oX 1225
```

overrides **daemon_smtp_ports**, but leaves **local_interfaces** unchanged, whereas

```
-oX 192.168.34.5.1125
```

overrides **local_interfaces**, leaving **daemon_smtp_ports** unchanged. (However, since **local_interfaces** now contains no items without ports, the value of **daemon_smtp_ports** is no longer relevant in this example.)

13.4 Support for the obsolete SSMTP (or SMTPS) protocol

Exim supports the obsolete SSMTP protocol (also known as SMTPS) that was used before the STARTTLS command was standardized for SMTP. Some legacy clients still use this protocol. If the **tls_on_connect_ports** option is set to a list of port numbers, connections to those ports must use SSMTP. The most common use of this option is expected to be

```
tls_on_connect_ports = 465
```

because 465 is the usual port number used by the legacy clients. There is also a command line option **-tls-on-connect**, which forces all ports to behave in this way when a daemon is started.

Warning: Setting **tls_on_connect_ports** does not of itself cause the daemon to listen on those ports. You must still specify them in **daemon_smtp_ports**, **local_interfaces**, or the **-oX** option. (This is

because **tls_on_connect_ports** applies to **inetd** connections as well as to connections via the daemon.)

13.5 IPv6 address scopes

IPv6 addresses have "scopes", and a host with multiple hardware interfaces can, in principle, have the same link-local IPv6 address on different interfaces. Thus, additional information is needed, over and above the IP address, to distinguish individual interfaces. A convention of using a percent sign followed by something (often the interface name) has been adopted in some cases, leading to addresses like this:

```
fe80::202:b3ff:fe03:45c1%eth0
```

To accommodate this usage, a percent sign followed by an arbitrary string is allowed at the end of an IPv6 address. By default, Exim calls *getaddrinfo()* to convert a textual IPv6 address for actual use. This function recognizes the percent convention in operating systems that support it, and it processes the address appropriately. Unfortunately, some older libraries have problems with *getaddrinfo()*. If

```
IPV6_USE_INET_PTON=yes
```

is set in *Local/Makefile* (or an OS-dependent Makefile) when Exim is built, Exim uses *inet_pton()* to convert a textual IPv6 address for actual use, instead of *getaddrinfo()*. (Before version 4.14, it always used this function.) Of course, this means that the additional functionality of *getaddrinfo()* – recognizing scoped addresses – is lost.

13.6 Disabling IPv6

Sometimes it happens that an Exim binary that was compiled with IPv6 support is run on a host whose kernel does not support IPv6. The binary will fall back to using IPv4, but it may waste resources looking up AAAA records, and trying to connect to IPv6 addresses, causing delays to mail delivery. If you set the **disable_ipv6** option true, even if the Exim binary has IPv6 support, no IPv6 activities take place. AAAA records are never looked up, and any IPv6 addresses that are listed in **local_interfaces**, data for the *manualroute* router, etc. are ignored. If IP literals are enabled, the *ipliteral* router declines to handle IPv6 literal addresses.

On the other hand, when IPv6 is in use, there may be times when you want to disable it for certain hosts or domains. You can use the **dns_ipv4_lookup** option to globally suppress the lookup of AAAA records for specified domains, and you can use the **ignore_target_hosts** generic router option to ignore IPv6 addresses in an individual router.

13.7 Examples of starting a listening daemon

The default case in an IPv6 environment is

```
daemon_smtp_ports = smtp
local_interfaces = <; ::0 ; 0.0.0.0
```

This specifies listening on the smtp port on all IPv6 and IPv4 interfaces. Either one or two sockets may be used, depending on the characteristics of the TCP/IP stack. (This is complicated and messy; for more information, read the comments in the *daemon.c* source file.)

To specify listening on ports 25 and 26 on all interfaces:

```
daemon_smtp_ports = 25 : 26
```

(leaving **local_interfaces** at the default setting) or, more explicitly:

```
local_interfaces = <; ::0.25     ; ::0.26 \
                      0.0.0.0.25 ; 0.0.0.0.26
```

To listen on the default port on all IPv4 interfaces, and on port 26 on the IPv4 loopback address only:

```
local_interfaces = 0.0.0.0 : 127.0.0.1.26
```

To specify listening on the default port on specific interfaces only:

```
local_interfaces = 192.168.34.67 : 192.168.34.67
```

Warning: Such a setting excludes listening on the loopback interfaces.

13.8 Recognizing the local host

The **local_interfaces** option is also used when Exim needs to determine whether or not an IP address refers to the local host. That is, the IP addresses of all the interfaces on which a daemon is listening are always treated as local.

For this usage, port numbers in **local_interfaces** are ignored. If either of the items 0.0.0.0 or ::0 are encountered, Exim gets a complete list of available interfaces from the operating system, and extracts the relevant (that is, IPv4 or IPv6) addresses to use for checking.

Some systems set up large numbers of virtual interfaces in order to provide many virtual web servers. In this situation, you may want to listen for email on only a few of the available interfaces, but nevertheless treat all interfaces as local when routing. You can do this by setting **extra_local_interfaces** to a list of IP addresses, possibly including the "all" wildcard values. These addresses are recognized as local, but are not used for listening. Consider this example:

```
local_interfaces = <; 127.0.0.1 ; ::1 ; \
                      192.168.53.235 ; \
                      3ffe:2101:12:1:a00:20ff:fe86:a061

extra_local_interfaces = <; ::0 ; 0.0.0.0
```

The daemon listens on the loopback interfaces and just one IPv4 and one IPv6 address, but all available interface addresses are treated as local when Exim is routing.

In some environments the local host name may be in an MX list, but with an IP address that is not assigned to any local interface. In other cases it may be desirable to treat other host names as if they referred to the local host. Both these cases can be handled by setting the **hosts_treat_as_local** option. This contains host names rather than IP addresses. When a host is referenced during routing, either via an MX record or directly, it is treated as the local host if its name matches **hosts_treat_as_local**, or if any of its IP addresses match **local_interfaces** or **extra_local_interfaces**.

13.9 Delivering to a remote host

Delivery to a remote host is handled by the smtp transport. By default, it allows the system's TCP/IP functions to choose which interface to use (if there is more than one) when connecting to a remote host. However, the **interface** option can be set to specify which interface is used. See the description of the smtp transport in chapter 30 for more details.

14. Main configuration

The first part of the run time configuration file contains three types of item:

- Macro definitions: These lines start with an upper case letter. See section 6.4 for details of macro processing.
- Named list definitions: These lines start with one of the words “domainlist”, “hostlist”, “addresslist”, or “localpartlist”. Their use is described in section 10.5.
- Main configuration settings: Each setting occupies one line of the file (with possible continuations). If any setting is preceded by the word “hide”, the **-bP** command line option displays its value to admin users only. See section 6.10 for a description of the syntax of these option settings.

This chapter specifies all the main configuration options, along with their types and default values. For ease of finding a particular option, they appear in alphabetical order in section 14.23 below. However, because there are now so many options, they are first listed briefly in functional groups, as an aid to finding the name of the option you are looking for. Some options are listed in more than one group.

14.1 Miscellaneous

bi_command	to run for **-bi** command line option
disable_ipv6	do no IPv6 processing
keep_malformed	for broken files – should not happen
localhost_number	for unique message ids in clusters
message_body_newlines	retain newlines in *$message_body*
message_body_visible	how much to show in *$message_body*
mua_wrapper	run in “MUA wrapper” mode
print_topbitchars	top-bit characters are printing
timezone	force time zone

14.2 Exim parameters

exim_group	override compiled-in value
exim_path	override compiled-in value
exim_user	override compiled-in value
primary_hostname	default from *uname()*
split_spool_directory	use multiple directories
spool_directory	override compiled-in value

14.3 Privilege controls

admin_groups	groups that are Exim admin users
deliver_drop_privilege	drop root for delivery processes
local_from_check	insert *Sender:* if necessary
local_from_prefix	for testing *From:* for local sender
local_from_suffix	for testing *From:* for local sender
local_sender_retain	keep *Sender:* from untrusted user
never_users	do not run deliveries as these
prod_requires_admin	forced delivery requires admin user
queue_list_requires_admin	queue listing requires admin user
trusted_groups	groups that are trusted
trusted_users	users that are trusted

14.4 Logging

hosts_connection_nolog	exemption from connect logging
log_file_path	override compiled-in value

log_selector	set/unset optional logging
log_timezone	add timezone to log lines
message_logs	create per-message logs
preserve_message_logs	after message completion
process_log_path	for SIGUSR1 and *exiwhat*
syslog_duplication	controls duplicate log lines on syslog
syslog_facility	set syslog “facility” field
syslog_processname	set syslog “ident” field
syslog_timestamp	timestamp syslog lines
write_rejectlog	control use of message log

14.5 Frozen messages

auto_thaw	sets time for retrying frozen messages
freeze_tell	send message when freezing
move_frozen_messages	to another directory
timeout_frozen_after	keep frozen messages only so long

14.6 Data lookups

ibase_servers	InterBase servers
ldap_default_servers	used if no server in query
ldap_version	set protocol version
lookup_open_max	lookup files held open
mysql_servers	default MySQL servers
oracle_servers	Oracle servers
pgsql_servers	default PostgreSQL servers
sqlite_lock_timeout	as it says

14.7 Message ids

message_id_header_domain	used to build *Message-ID:* header
message_id_header_text	ditto

14.8 Embedded Perl Startup

perl_at_start	always start the interpreter
perl_startup	code to obey when starting Perl

14.9 Daemon

daemon_smtp_ports	default ports
daemon_startup_retries	number of times to retry
daemon_startup_sleep	time to sleep between tries
extra_local_interfaces	not necessarily listened on
local_interfaces	on which to listen, with optional ports
pid_file_path	override compiled-in value
queue_run_max	maximum simultaneous queue runners

14.10 Resource control

check_log_inodes	before accepting a message
check_log_space	before accepting a message
check_spool_inodes	before accepting a message
check_spool_space	before accepting a message
deliver_queue_load_max	no queue deliveries if load high
queue_only_load	queue incoming if load high
queue_only_load_latch	don’t re-evaluate load for each message

queue_run_max	maximum simultaneous queue runners
remote_max_parallel	parallel SMTP delivery per message
smtp_accept_max	simultaneous incoming connections
smtp_accept_max_nonmail	non-mail commands
smtp_accept_max_nonmail_hosts	hosts to which the limit applies
smtp_accept_max_per_connection	messages per connection
smtp_accept_max_per_host	connections from one host
smtp_accept_queue	queue mail if more connections
smtp_accept_queue_per_connection	queue if more messages per connection
smtp_accept_reserve	only reserve hosts if more connections
smtp_check_spool_space	from SIZE on MAIL command
smtp_connect_backlog	passed to TCP/IP stack
smtp_load_reserve	SMTP from reserved hosts if load high
smtp_reserve_hosts	these are the reserve hosts

14.11 Policy controls

acl_not_smtp	ACL for non-SMTP messages
acl_not_smtp_mime	ACL for non-SMTP MIME parts
acl_not_smtp_start	ACL for start of non-SMTP message
acl_smtp_auth	ACL for AUTH
acl_smtp_connect	ACL for connection
acl_smtp_data	ACL for DATA
acl_smtp_etrn	ACL for ETRN
acl_smtp_expn	ACL for EXPN
acl_smtp_helo	ACL for EHLO or HELO
acl_smtp_mail	ACL for MAIL
acl_smtp_mailauth	ACL for AUTH on MAIL command
acl_smtp_mime	ACL for MIME parts
acl_smtp_predata	ACL for start of data
acl_smtp_quit	ACL for QUIT
acl_smtp_rcpt	ACL for RCPT
acl_smtp_starttls	ACL for STARTTLS
acl_smtp_vrfy	ACL for VRFY
av_scanner	specify virus scanner
check_rfc2047_length	check length of RFC 2047 “encoded words”
dns_csa_search_limit	control CSA parent search depth
dns_csa_use_reverse	en/disable CSA IP reverse search
header_maxsize	total size of message header
header_line_maxsize	individual header line limit
helo_accept_junk_hosts	allow syntactic junk from these hosts
helo_allow_chars	allow illegal chars in HELO names
helo_lookup_domains	lookup hostname for these HELO names
helo_try_verify_hosts	HELO soft-checked for these hosts
helo_verify_hosts	HELO hard-checked for these hosts
host_lookup	host name looked up for these hosts
host_lookup_order	order of DNS and local name lookups
host_reject_connection	reject connection from these hosts
hosts_treat_as_local	useful in some cluster configurations
local_scan_timeout	timeout for *local_scan()*
message_size_limit	for all messages
percent_hack_domains	recognize %-hack for these domains
spamd_address	set interface to SpamAssassin
strict_acl_vars	object to unset ACL variables

14.12 Callout cache

callout_domain_negative_expire	timeout for negative domain cache item
callout_domain_positive_expire	timeout for positive domain cache item
callout_negative_expire	timeout for negative address cache item
callout_positive_expire	timeout for positive address cache item
callout_random_local_part	string to use for "random" testing

14.13 TLS

gnutls_require_kx	control GnuTLS key exchanges
gnutls_require_mac	control GnuTLS MAC algorithms
gnutls_require_protocols	control GnuTLS protocols
tls_advertise_hosts	advertise TLS to these hosts
tls_certificate	location of server certificate
tls_crl	certificate revocation list
tls_dhparam	DH parameters for server
tls_on_connect_ports	specify SSMTP (SMTPS) ports
tls_privatekey	location of server private key
tls_remember_esmtp	don't reset after starting TLS
tls_require_ciphers	specify acceptable ciphers
tls_try_verify_hosts	try to verify client certificate
tls_verify_certificates	expected client certificates
tls_verify_hosts	insist on client certificate verify

14.14 Local user handling

finduser_retries	useful in NIS environments
gecos_name	used when creating *Sender:*
gecos_pattern	ditto
max_username_length	for systems that truncate
unknown_login	used when no login name found
unknown_username	ditto
uucp_from_pattern	for recognizing "From " lines
uucp_from_sender	ditto

14.15 All incoming messages (SMTP and non-SMTP)

header_maxsize	total size of message header
header_line_maxsize	individual header line limit
message_size_limit	applies to all messages
percent_hack_domains	recognize %-hack for these domains
received_header_text	expanded to make *Received:*
received_headers_max	for mail loop detection
recipients_max	limit per message
recipients_max_reject	permanently reject excess recipients

14.16 Non-SMTP incoming messages

receive_timeout	for non-SMTP messages

14.17 Incoming SMTP messages

See also the *Policy controls* section above.

host_lookup	host name looked up for these hosts
host_lookup_order	order of DNS and local name lookups
recipient_unqualified_hosts	may send unqualified recipients
rfc1413_hosts	make ident calls to these hosts

rfc1413_query_timeout	zero disables ident calls
sender_unqualified_hosts	may send unqualified senders
smtp_accept_keepalive	some TCP/IP magic
smtp_accept_max	simultaneous incoming connections
smtp_accept_max_nonmail	non-mail commands
smtp_accept_max_nonmail_hosts	hosts to which the limit applies
smtp_accept_max_per_connection	messages per connection
smtp_accept_max_per_host	connections from one host
smtp_accept_queue	queue mail if more connections
smtp_accept_queue_per_connection	queue if more messages per connection
smtp_accept_reserve	only reserve hosts if more connections
smtp_active_hostname	host name to use in messages
smtp_banner	text for welcome banner
smtp_check_spool_space	from SIZE on MAIL command
smtp_connect_backlog	passed to TCP/IP stack
smtp_enforce_sync	of SMTP command/responses
smtp_etrn_command	what to run for ETRN
smtp_etrn_serialize	only one at once
smtp_load_reserve	only reserve hosts if this load
smtp_max_unknown_commands	before dropping connection
smtp_ratelimit_hosts	apply ratelimiting to these hosts
smtp_ratelimit_mail	ratelimit for MAIL commands
smtp_ratelimit_rcpt	ratelimit for RCPT commands
smtp_receive_timeout	per command or data line
smtp_reserve_hosts	these are the reserve hosts
smtp_return_error_details	give detail on rejections

14.18 SMTP extensions

accept_8bitmime	advertise 8BITMIME
auth_advertise_hosts	advertise AUTH to these hosts
ignore_fromline_hosts	allow “From ” from these hosts
ignore_fromline_local	allow “From ” from local SMTP
pipelining_advertise_hosts	advertise pipelining to these hosts
tls_advertise_hosts	advertise TLS to these hosts

14.19 Processing messages

allow_domain_literals	recognize domain literal syntax
allow_mx_to_ip	allow MX to point to IP address
allow_utf8_domains	in addresses
check_rfc2047_length	check length of RFC 2047 “encoded words”
delivery_date_remove	from incoming messages
envelope_to_remove	from incoming messages
extract_addresses_remove_arguments	affects **-t** processing
headers_charset	default for translations
qualify_domain	default for senders
qualify_recipient	default for recipients
return_path_remove	from incoming messages
strip_excess_angle_brackets	in addresses
strip_trailing_dot	at end of addresses
untrusted_set_sender	untrusted can set envelope sender

14.20 System filter

system_filter	locate system filter
system_filter_directory_transport	transport for delivery to a directory
system_filter_file_transport	transport for delivery to a file

system_filter_group	group for filter running
system_filter_pipe_transport	transport for delivery to a pipe
system_filter_reply_transport	transport for autoreply delivery
system_filter_user	user for filter running

14.21 Routing and delivery

disable_ipv6	do no IPv6 processing
dns_again_means_nonexist	for broken domains
dns_check_names_pattern	pre-DNS syntax check
dns_ipv4_lookup	only v4 lookup for these domains
dns_retrans	parameter for resolver
dns_retry	parameter for resolver
hold_domains	hold delivery for these domains
local_interfaces	for routing checks
queue_domains	no immediate delivery for these
queue_only	no immediate delivery at all
queue_only_file	no immediate delivery if file exists
queue_only_load	no immediate delivery if load is high
queue_only_load_latch	don't re-evaluate load for each message
queue_only_override	allow command line to override
queue_run_in_order	order of arrival
queue_run_max	of simultaneous queue runners
queue_smtp_domains	no immediate SMTP delivery for these
remote_max_parallel	parallel SMTP delivery per message
remote_sort_domains	order of remote deliveries
retry_data_expire	timeout for retry data
retry_interval_max	safety net for retry rules

14.22 Bounce and warning messages

bounce_message_file	content of bounce
bounce_message_text	content of bounce
bounce_return_body	include body if returning message
bounce_return_message	include original message in bounce
bounce_return_size_limit	limit on returned message
bounce_sender_authentication	send authenticated sender with bounce
dsn_from	set *From:* contents in bounces
errors_copy	copy bounce messages
errors_reply_to	*Reply-to:* in bounces
delay_warning	time schedule
delay_warning_condition	condition for warning messages
ignore_bounce_errors_after	discard undeliverable bounces
smtp_return_error_details	give detail on rejections
warn_message_file	content of warning message

14.23 Alphabetical list of main options

Those options that undergo string expansion before use are marked with †.

accept_8bitmime	Use: *main*	Type: *boolean*	Default: *false*

This option causes Exim to send 8BITMIME in its response to an SMTP EHLO command, and to accept the BODY= parameter on MAIL commands. However, though Exim is 8-bit clean, it is not a protocol converter, and it takes no steps to do anything special with messages received by this route. Consequently, this option is turned off by default.

acl_not_smtp	Use: *main*	Type: *string*†	Default: *unset*

This option defines the ACL that is run when a non-SMTP message has been read and is on the point of being accepted. See chapter 40 for further details.

acl_not_smtp_mime	Use: *main*	Type: *string*†	Default: *unset*

This option defines the ACL that is run for individual MIME parts of non-SMTP messages. It operates in exactly the same way as **acl_smtp_mime** operates for SMTP messages.

acl_not_smtp_start	Use: *main*	Type: *string*†	Default: *unset*

This option defines the ACL that is run before Exim starts reading a non-SMTP message. See chapter 40 for further details.

acl_smtp_auth	Use: *main*	Type: *string*†	Default: *unset*

This option defines the ACL that is run when an SMTP AUTH command is received. See chapter 40 for further details.

acl_smtp_connect	Use: *main*	Type: *string*†	Default: *unset*

This option defines the ACL that is run when an SMTP connection is received. See chapter 40 for further details.

acl_smtp_data	Use: *main*	Type: *string*†	Default: *unset*

This option defines the ACL that is run after an SMTP DATA command has been processed and the message itself has been received, but before the final acknowledgment is sent. See chapter 40 for further details.

acl_smtp_etrn	Use: *main*	Type: *string*†	Default: *unset*

This option defines the ACL that is run when an SMTP ETRN command is received. See chapter 40 for further details.

acl_smtp_expn	Use: *main*	Type: *string*†	Default: *unset*

This option defines the ACL that is run when an SMTP EXPN command is received. See chapter 40 for further details.

acl_smtp_helo	Use: *main*	Type: *string*†	Default: *unset*

This option defines the ACL that is run when an SMTP EHLO or HELO command is received. See chapter 40 for further details.

acl_smtp_mail	Use: *main*	Type: *string*†	Default: *unset*

This option defines the ACL that is run when an SMTP MAIL command is received. See chapter 40 for further details.

acl_smtp_mailauth	Use: *main*	Type: *string*†	Default: *unset*

This option defines the ACL that is run when there is an AUTH parameter on a MAIL command. See chapter 40 for details of ACLs, and chapter 33 for details of authentication.

acl_smtp_mime	Use: *main*	Type: *string*†	Default: *unset*

This option is available when Exim is built with the content-scanning extension. It defines the ACL that is run for each MIME part in a message. See section 41.4 for details.

acl_smtp_predata	Use: *main*	Type: *string*†	Default: *unset*

This option defines the ACL that is run when an SMTP DATA command is received, before the message itself is received. See chapter 40 for further details.

acl_smtp_quit	Use: *main*	Type: *string*†	Default: *unset*

This option defines the ACL that is run when an SMTP QUIT command is received. See chapter 40 for further details.

acl_smtp_rcpt	Use: *main*	Type: *string*†	Default: *unset*

This option defines the ACL that is run when an SMTP RCPT command is received. See chapter 40 for further details.

acl_smtp_starttls	Use: *main*	Type: *string*†	Default: *unset*

This option defines the ACL that is run when an SMTP STARTTLS command is received. See chapter 40 for further details.

acl_smtp_vrfy	Use: *main*	Type: *string*†	Default: *unset*

This option defines the ACL that is run when an SMTP VRFY command is received. See chapter 40 for further details.

admin_groups	Use: *main*	Type: *string list*†	Default: *unset*

This option is expanded just once, at the start of Exim's processing. If the current group or any of the supplementary groups of an Exim caller is in this colon-separated list, the caller has admin privileges. If all your system programmers are in a specific group, for example, you can give them all Exim admin privileges by putting that group in **admin_groups**. However, this does not permit them to read Exim's spool files (whose group owner is the Exim gid). To permit this, you have to add individuals to the Exim group.

allow_domain_literals	Use: *main*	Type: *boolean*	Default: *false*

If this option is set, the RFC 2822 domain literal format is permitted in email addresses. The option is not set by default, because the domain literal format is not normally required these days, and few people know about it. It has, however, been exploited by mail abusers.

Unfortunately, it seems that some DNS black list maintainers are using this format to report black listing to postmasters. If you want to accept messages addressed to your hosts by IP address, you need to set **allow_domain_literals** true, and also to add `@[]` to the list of local domains (defined in the

named domain list **local_domains** in the default configuration). This "magic string" matches the domain literal form of all the local host's IP addresses.

allow_mx_to_ip	Use: *main*	Type: *boolean*	Default: *false*

It appears that more and more DNS zone administrators are breaking the rules and putting domain names that look like IP addresses on the right hand side of MX records. Exim follows the rules and rejects this, giving an error message that explains the mis-configuration. However, some other MTAs support this practice, so to avoid "Why can't Exim do this?" complaints, **allow_mx_to_ip** exists, in order to enable this heinous activity. It is not recommended, except when you have no other choice.

allow_utf8_domains	Use: *main*	Type: *boolean*	Default: *false*

Lots of discussion is going on about internationalized domain names. One camp is strongly in favour of just using UTF-8 characters, and it seems that at least two other MTAs permit this. This option allows Exim users to experiment if they wish.

If it is set true, Exim's domain parsing function allows valid UTF-8 multicharacters to appear in domain name components, in addition to letters, digits, and hyphens. However, just setting this option is not enough; if you want to look up these domain names in the DNS, you must also adjust the value of **dns_check_names_pattern** to match the extended form. A suitable setting is:

```
dns_check_names_pattern = (?i)^(?>(?(1)\.|())[a-z0-9\xc0-\xff]\
  (?>[-a-z0-9\x80-\xff]*[a-z0-9\x80-\xbf])?)+$
```

Alternatively, you can just disable this feature by setting

```
dns_check_names_pattern =
```

That is, set the option to an empty string so that no check is done.

auth_advertise_hosts	Use: *main*	Type: *host list*†	Default: *

If any server authentication mechanisms are configured, Exim advertises them in response to an EHLO command only if the calling host matches this list. Otherwise, Exim does not advertise AUTH. Exim does not accept AUTH commands from clients to which it has not advertised the availability of AUTH. The advertising of individual authentication mechanisms can be controlled by the use of the **server_advertise_condition** generic authenticator option on the individual authenticators. See chapter 33 for further details.

Certain mail clients (for example, Netscape) require the user to provide a name and password for authentication if AUTH is advertised, even though it may not be needed (the host may accept messages from hosts on its local LAN without authentication, for example). The **auth_advertise_hosts** option can be used to make these clients more friendly by excluding them from the set of hosts to which Exim advertises AUTH.

If you want to advertise the availability of AUTH only when the connection is encrypted using TLS, you can make use of the fact that the value of this option is expanded, with a setting like this:

```
auth_advertise_hosts = ${if eq{$tls_cipher}{}{}{*}}
```

If *$tls_cipher* is empty, the session is not encrypted, and the result of the expansion is empty, thus matching no hosts. Otherwise, the result of the expansion is *, which matches all hosts.

auto_thaw	Use: *main*	Type: *time*	Default: *0s*

If this option is set to a time greater than zero, a queue runner will try a new delivery attempt on any frozen message, other than a bounce message, if this much time has passed since it was frozen. This may result in the message being re-frozen if nothing has changed since the last attempt. It is a way of saying "keep on trying, even though there are big problems".

Note: This is an old option, which predates **timeout_frozen_after** and **ignore_bounce_errors_after**. It is retained for compatibility, but it is not thought to be very useful any more, and its use should probably be avoided.

av_scanner	Use: *main*	Type: *string*	Default: *see below*

This option is available if Exim is built with the content-scanning extension. It specifies which anti-virus scanner to use. The default value is:

```
sophie:/var/run/sophie
```

If the value of **av_scanner** starts with dollar character, it is expanded before use. See section 41.1 for further details.

bi_command	Use: *main*	Type: *string*	Default: *unset*

This option supplies the name of a command that is run when Exim is called with the **-bi** option (see chapter 5). The string value is just the command name, it is not a complete command line. If an argument is required, it must come from the **-oA** command line option.

bounce_message_file	Use: *main*	Type: *string*	Default: *unset*

This option defines a template file containing paragraphs of text to be used for constructing bounce messages. Details of the file's contents are given in chapter 46. See also **warn_message_file**.

bounce_message_text	Use: *main*	Type: *string*	Default: *unset*

When this option is set, its contents are included in the default bounce message immediately after "This message was created automatically by mail delivery software." It is not used if **bounce_message_file** is set.

bounce_return_body	Use: *main*	Type: *boolean*	Default: *true*

This option controls whether the body of an incoming message is included in a bounce message when **bounce_return_message** is true. The default setting causes the entire message, both header and body, to be returned (subject to the value of **bounce_return_size_limit**). If this option is false, only the message header is included. In the case of a non-SMTP message containing an error that is detected during reception, only those header lines preceding the point at which the error was detected are returned.

bounce_return_message	Use: *main*	Type: *boolean*	Default: *true*

If this option is set false, none of the original message is included in bounce messages generated by Exim. See also **bounce_return_size_limit** and **bounce_return_body**.

bounce_return_size_limit	Use: *main*	Type: *integer*	Default: *100K*

This option sets a limit in bytes on the size of messages that are returned to senders as part of bounce messages when **bounce_return_message** is true. The limit should be less than the value of the global **message_size_limit** and of any **message_size_limit** settings on transports, to allow for the bounce text that Exim generates. If this option is set to zero there is no limit.

When the body of any message that is to be included in a bounce message is greater than the limit, it is truncated, and a comment pointing this out is added at the top. The actual cutoff may be greater

than the value given, owing to the use of buffering for transferring the message in chunks (typically 8K in size). The idea is to save bandwidth on those undeliverable 15-megabyte messages.

bounce_sender_authentication	Use: *main*	Type: *string*	Default: *unset*

This option provides an authenticated sender address that is sent with any bounce messages generated by Exim that are sent over an authenticated SMTP connection. A typical setting might be:

```
bounce_sender_authentication = mailer-daemon@my.domain.example
```

which would cause bounce messages to be sent using the SMTP command:

```
MAIL FROM:<> AUTH=mailer-daemon@my.domain.example
```

The value of **bounce_sender_authentication** must always be a complete email address.

callout_domain_negative_expire	Use: *main*	Type: *time*	Default: *3h*

This option specifies the expiry time for negative callout cache data for a domain. See section 40.41 for details of callout verification, and section 40.43 for details of the caching.

callout_domain_positive_expire	Use: *main*	Type: *time*	Default: *7d*

This option specifies the expiry time for positive callout cache data for a domain. See section 40.41 for details of callout verification, and section 40.43 for details of the caching.

callout_negative_expire	Use: *main*	Type: *time*	Default: *2h*

This option specifies the expiry time for negative callout cache data for an address. See section 40.41 for details of callout verification, and section 40.43 for details of the caching.

callout_positive_expire	Use: *main*	Type: *time*	Default: *24h*

This option specifies the expiry time for positive callout cache data for an address. See section 40.41 for details of callout verification, and section 40.43 for details of the caching.

callout_random_local_part	Use: *main*	Type: *string*†	Default: *see below*

This option defines the "random" local part that can be used as part of callout verification. The default value is

```
$primary_host_name-$tod_epoch-testing
```

See section 40.42 for details of how this value is used.

check_log_inodes	Use: *main*	Type: *integer*	Default: *0*

See **check_spool_space** below.

check_log_space	Use: *main*	Type: *integer*	Default: *0*

See **check_spool_space** below.

check_rfc2047_length	Use: *main*	Type: *boolean*	Default: *true*

RFC 2047 defines a way of encoding non-ASCII characters in headers using a system of "encoded words". The RFC specifies a maximum length for an encoded word; strings to be encoded that exceed this length are supposed to use multiple encoded words. By default, Exim does not recognize encoded words that exceed the maximum length. However, it seems that some software, in violation of the RFC, generates overlong encoded words. If **check_rfc2047_length** is set false, Exim recognizes encoded words of any length.

check_spool_inodes	Use: *main*	Type: *integer*	Default: *0*

See **check_spool_space** below.

check_spool_space	Use: *main*	Type: *integer*	Default: *0*

The four **check_...** options allow for checking of disk resources before a message is accepted.

When any of these options are set, they apply to all incoming messages. If you want to apply different checks to different kinds of message, you can do so by testing the variables *$log_inodes*, *$log_space*, *$spool_inodes*, and *$spool_space* in an ACL with appropriate additional conditions.

check_spool_space and **check_spool_inodes** check the spool partition if either value is greater than zero, for example:

```
check_spool_space = 10M
check_spool_inodes = 100
```

The spool partition is the one that contains the directory defined by SPOOL_DIRECTORY in *Local/Makefile*. It is used for holding messages in transit.

check_log_space and **check_log_inodes** check the partition in which log files are written if either is greater than zero. These should be set only if **log_file_path** and **spool_directory** refer to different partitions.

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 ESMTP is in use and there was 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.

The values for **check_spool_space** and **check_log_space** are held as a number of kilobytes. If a non-multiple of 1024 is specified, it is rounded up.

For non-SMTP input and for batched SMTP input, the test is done at start-up; on failure a message is written to stderr and Exim exits with a non-zero code, as it obviously cannot send an error message of any kind.

daemon_smtp_ports	Use: *main*	Type: *string*	Default: `smtp`

This option specifies one or more default SMTP ports on which the Exim daemon listens. See chapter 13 for details of how it is used. For backward compatibility, **daemon_smtp_port** (singular) is a synonym.

daemon_startup_retries	Use: *main*	Type: *integer*	Default: *9*

This option, along with **daemon_startup_sleep**, controls the retrying done by the daemon at startup when it cannot immediately bind a listening socket (typically because the socket is already in use): **daemon_startup_retries** defines the number of retries after the first failure, and **daemon_startup_sleep** defines the length of time to wait between retries.

daemon_startup_sleep	Use: *main*	Type: *time*	Default: *30s*

See **daemon_startup_retries**.

delay_warning	Use: *main*	Type: *time list*	Default: *24h*

When a message is delayed, Exim sends a warning message to the sender at intervals specified by this option. The data is a colon-separated list of times after which to send warning messages. If the value of the option is an empty string or a zero time, no warnings are sent. Up to 10 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 = 4h:8h:24h
```

the first message is sent after 4 hours, the second after 8 hours, and the third one after 24 hours. After that, messages are sent every 16 hours, because that is the interval between the last two times on the list. If you set just one time, it specifies the repeat interval. For example, with:

```
delay_warning = 6h
```

messages are repeated every six hours. To stop warnings after a given time, set a very large time at the end of the list. For example:

```
delay_warning = 2h:12h:99d
```

delay_warning_condition	Use: *main*	Type: *string*†	Default: *see below*

The 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) then the warning message is not sent. The default is:

```
delay_warning_condition = ${if or {\
  { !eq{$h_list-id:$h_list-post:$h_list-subscribe:}{} }\
  { match{$h_precedence:}{(?i)bulk|list|junk} }\
  { match{$h_auto-submitted:}{(?i)auto-generated|auto-replied} }\
  } {no}{yes}}
```

This suppresses the sending of warnings for messages that contain *List-ID:*, *List-Post:*, or *List-Subscribe:* headers, or have "bulk", "list" or "junk" in a *Precedence:* header, or have "auto-generated" or "auto-replied" in an *Auto-Submitted:* header.

deliver_drop_privilege	Use: *main*	Type: *boolean*	Default: *false*

If this option is set true, Exim drops its root privilege at the start of a delivery process, and runs as the Exim user throughout. This severely restricts the kinds of local delivery that are possible, but is viable in certain types of configuration. There is a discussion about the use of root privilege in chapter 52.

deliver_queue_load_max	Use: *main*	Type: *fixed-point*	Default: *unset*

When this option is set, a queue run is abandoned if the system load average becomes greater than the value of the option. The option has no effect on ancient operating systems on which Exim cannot determine the load average. See also **queue_only_load** and **smtp_load_reserve**.

delivery_date_remove	Use: *main*	Type: *boolean*	Default: *true*

Exim's transports have an option for adding a *Delivery-date:* header to a message when it is delivered, in exactly the same way as *Return-path:* is handled. *Delivery-date:* records the actual time of delivery.

Such headers should not be present in incoming messages, and this option causes them to be removed at the time the message is received, to avoid any problems that might occur when a delivered message is subsequently sent on to some other recipient.

disable_fsync	Use: *main*	Type: *boolean*	Default: *false*

This option is available only if Exim was built with the compile-time option ENABLE_DISABLE_FSYNC. When this is not set, a reference to **disable_fsync** in a runtime configuration generates an "unknown option" error. You should not build Exim with ENABLE_DISABLE_FSYNC or set **disable_fsync** unless you really, really, really understand what you are doing. *No pre-compiled distributions of Exim should ever make this option available.*

When **disable_fsync** is set true, Exim no longer calls *fsync()* to force updated files' data to be written to disc before continuing. Unexpected events such as crashes and power outages may cause data to be lost or scrambled. Here be Dragons. **Beware.**

disable_ipv6	Use: *main*	Type: *boolean*	Default: *false*

If this option is set true, even if the Exim binary has IPv6 support, no IPv6 activities take place. AAAA records are never looked up, and any IPv6 addresses that are listed in **local_interfaces**, data for the **manualroute** router, etc. are ignored. If IP literals are enabled, the *ipliteral* router declines to handle IPv6 literal addresses.

dns_again_means_nonexist	Use: *main*	Type: *domain list*†	Default: *unset*

DNS lookups give a "try again" response for the DNS errors "non-authoritative host not found" and "SERVERFAIL". This can cause Exim to keep trying to deliver a message, or to give repeated temporary errors to incoming mail. Sometimes the effect is caused by a badly set up name server and may persist for a long time. If a domain which exhibits this problem matches anything in **dns_again_means_nonexist**, it is treated as if it did not exist. This option should be used with care. You can make it apply to reverse lookups by a setting such as this:

```
dns_again_means_nonexist = *.in-addr.arpa
```

This option applies to all DNS lookups that Exim does. It also applies when the *gethostbyname()* or *getipnodebyname()* functions give temporary errors, since these are most likely to be caused by DNS lookup problems. The *dnslookup* router has some options of its own for controlling what happens when lookups for MX or SRV records give temporary errors. These more specific options are applied after this global option.

dns_check_names_pattern	Use: *main*	Type: *string*	Default: *see below*

When this option is set to a non-empty string, it causes Exim to check domain names for characters that are not allowed in host names before handing them to the DNS resolver, because some resolvers give temporary errors for names that contain unusual characters. If a domain name contains any unwanted characters, a "not found" result is forced, and the resolver is not called. The check is done by matching the domain name against a regular expression, which is the value of this option. The default pattern is

```
dns_check_names_pattern = \
  (?i)^(?>(?(1)\.|())[^\W_](?>[a-z0-9/-]*[^\W_])?)+$
```

which permits only letters, digits, slashes, and hyphens in components, but they must start and end with a letter or digit. Slashes are not, in fact, permitted in host names, but they are found in certain NS records (which can be accessed in Exim by using a **dnsdb** lookup). If you set **allow_utf8_domains**, you must modify this pattern, or set the option to an empty string.

dns_csa_search_limit	Use: *main*	Type: *integer*	Default: *5*

This option controls the depth of parental searching for CSA SRV records in the DNS, as described in more detail in section 40.46.

dns_csa_use_reverse	Use: *main*	Type: *boolean*	Default: *true*

This option controls whether or not an IP address, given as a CSA domain, is reversed and looked up in the reverse DNS, as described in more detail in section 40.46.

dns_ipv4_lookup	Use: *main*	Type: *domain list*†	Default: *unset*

When Exim is compiled with IPv6 support and **disable_ipv6** is not set, it looks for IPv6 address records (AAAA records) as well as IPv4 address records (A records) when trying to find IP addresses for hosts, unless the host's domain matches this list.

This is a fudge to help with name servers that give big delays or otherwise do not work for the AAAA record type. In due course, when the world's name servers have all been upgraded, there should be no need for this option.

dns_retrans	Use: *main*	Type: *time*	Default: *0s*

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 first value is the time between retries, and the second is the number of retries. It isn't totally clear exactly how these settings affect the total time a DNS lookup may take. I haven't found any documentation about timeouts on DNS lookups; these parameter values are available in the external resolver interface structure, but nowhere does it seem to describe how they are used or what you might want to set in them.

dns_retry	Use: *main*	Type: *integer*	Default: *0*

See **dns_retrans** above.

drop_cr	Use: *main*	Type: *boolean*	Default: *false*

This is an obsolete option that is now a no-op. It used to affect the way Exim handled CR and LF characters in incoming messages. What happens now is described in section 44.2.

dsn_from	Use: *main*	Type: *string*†	Default: *see below*

This option can be used to vary the contents of *From:* header lines in bounces and other automatically generated messages ("Delivery Status Notifications" – hence the name of the option). The default setting is:

```
dsn_from = Mail Delivery System <Mailer-Daemon@$qualify_domain>
```

The value is expanded every time it is needed. If the expansion fails, a panic is logged, and the default value is used.

envelope_to_remove	Use: *main*	Type: *boolean*	Default: *true*

Exim's transports have an option for adding an *Envelope-to:* header to a message when it is delivered, in exactly the same way as *Return-path:* is handled. *Envelope-to:* records the original recipient address from the messages's envelope that caused the delivery to happen. Such headers should not be

present in incoming messages, and this option causes them to be removed at the time the message is received, to avoid any problems that might occur when a delivered message is subsequently sent on to some other recipient.

errors_copy	Use: *main*	Type: *string list*†	Default: *unset*

Setting this option causes Exim to send bcc copies of bounce messages that it generates to other addresses. **Note**: This does not apply to bounce messages coming from elsewhere. The value of the option is a colon-separated list of items. Each item consists of a pattern, terminated by white space, followed by a comma-separated list of email addresses. If a pattern contains spaces, it must be enclosed in double quotes.

Each pattern is processed in the same way as a single item in an address list (see section 10.19). When a pattern matches the recipient of the bounce message, the message is copied to the addresses on the list. The items are scanned in order, and once a matching one is found, no further items are examined. For example:

```
errors_copy = spqr@mydomain   postmaster@mydomain.example :\
              rqps@mydomain   hostmaster@mydomain.example,\
                              postmaster@mydomain.example
```

The address list is expanded before use. The expansion variables *$local_part* and *$domain* are set from the original recipient of the error message, and if there was any wildcard matching in the pattern, the expansion variables *$0*, *$1*, etc. are set in the normal way.

errors_reply_to	Use: *main*	Type: *string*	Default: *unset*

By default, Exim's bounce and delivery warning messages contain the header line

```
From: Mail Delivery System <Mailer-Daemon@qualify-domain>
```

where *qualify-domain* is the value of the **qualify_domain** option. A warning message that is generated by the **quota_warn_message** option in an *appendfile* transport may contain its own *From:* header line that overrides the default.

Experience shows that people reply to bounce messages. If the **errors_reply_to** option is set, a *Reply-To:* header is added to bounce and warning messages. For example:

```
errors_reply_to = postmaster@my.domain.example
```

The value of the option is not expanded. It must specify a valid RFC 2822 address. However, if a warning message that is generated by the **quota_warn_message** option in an *appendfile* transport contain its own *Reply-To:* header line, the value of the **errors_reply_to** option is not used.

exim_group	Use: *main*	Type: *string*	Default: *compile-time configured*

This option changes the gid under which Exim runs when it gives up root privilege. The default value is compiled into the binary. The value of this option is used only when **exim_user** is also set. Unless it consists entirely of digits, the string is looked up using *getgrnam()*, and failure causes a configuration error. See chapter 52 for a discussion of security issues.

exim_path	Use: *main*	Type: *string*	Default: *see below*

This option specifies the path name of the Exim binary, which is used when Exim needs to re-exec itself. The default is set up to point to the file *exim* in the directory configured at compile time by the BIN_DIRECTORY setting. It is necessary to change **exim_path** if, exceptionally, Exim is run from some other place. **Warning**: Do not use a macro to define the value of this option, because you will

break those Exim utilities that scan the configuration file to find where the binary is. (They then use the **-bP** option to extract option settings such as the value of **spool_directory**.)

exim_user	Use: *main*	Type: *string*	Default: *compile-time configured*

This option changes the uid under which Exim runs when it gives up root privilege. The default value is compiled into the binary. Ownership of the run time configuration file and the use of the **-C** and **-D** command line options is checked against the values in the binary, not what is set here.

Unless it consists entirely of digits, the string is looked up using *getpwnam()*, and failure causes a configuration error. If **exim_group** is not also supplied, the gid is taken from the result of *getpwnam()* if it is used. See chapter 52 for a discussion of security issues.

extra_local_interfaces	Use: *main*	Type: *string list*	Default: *unset*

This option defines network interfaces that are to be considered local when routing, but which are not used for listening by the daemon. See section 13.8 for details.

extract_addresses_remove_ arguments	Use: *main*	Type: *boolean*	Default: *true*

According to some Sendmail documentation (Sun, IRIX, HP-UX), if any addresses are present on the command line when the **-t** option is used to build an envelope from a message's *To:*, *Cc:* and *Bcc:* headers, the command line addresses are removed from the recipients list. This is also how Smail behaves. However, other Sendmail documentation (the O'Reilly book) states that command line addresses are added to those obtained from the header lines. When **extract_addresses_remove_ arguments** is true (the default), Exim subtracts argument headers. If it is set false, Exim adds rather than removes argument addresses.

finduser_retries	Use: *main*	Type: *integer*	Default: *0*

On systems running NIS or other schemes in which user and group information is distributed from a remote system, there can be times when *getpwnam()* and related functions fail, even when given valid data, because things time out. Unfortunately these failures cannot be distinguished from genuine "not found" errors. If **finduser_retries** is set greater than zero, Exim will try that many extra times to find a user or a group, waiting for one second between retries.

You should not set this option greater than zero if your user information is in a traditional */etc/passwd* file, because it will cause Exim needlessly to search the file multiple times for non-existent users, and also cause delay.

freeze_tell	Use: *main*	Type: *string list, comma separated*	Default: *unset*

On encountering certain errors, or when configured to do so in a system filter, ACL, or special router, Exim freezes a message. This means that no further delivery attempts take place until an administrator thaws the message, or the **auto_thaw**, **ignore_bounce_errors_after**, or **timeout_frozen_after** feature cause it to be processed. If **freeze_tell** is set, Exim generates a warning message whenever it freezes something, unless the message it is freezing is a locally-generated bounce message. (Without this exception there is the possibility of looping.) The warning message is sent to the addresses supplied as the comma-separated value of this option. If several of the message's addresses cause freezing, only a single message is sent. If the freezing was automatic, the reason(s) for freezing can be found in the message log. If you configure freezing in a filter or ACL, you must arrange for any logging that you require.

gecos_name	Use: *main*	Type: *string†*	Default: *unset*

Some operating systems, notably HP-UX, use the "gecos" field in the system password file to hold other information in addition to users' real names. Exim looks up this field for use when it is creating *Sender:* or *From:* headers. If either **gecos_pattern** or **gecos_name** are unset, the contents of the field are used unchanged, except that, if an ampersand is encountered, it is replaced by the user's login name with the first character forced to upper case, since this is a convention that is observed on many systems.

When these options are set, **gecos_pattern** is treated as a regular expression that is to be applied to the field (again with & replaced by the login name), and if it matches, **gecos_name** is expanded and used as the user's name.

Numeric variables such as *$1*, *$2*, etc. can be used in the expansion to pick up sub-fields that were matched by the pattern. In HP-UX, where the user's name terminates at the first comma, the following can be used:

```
gecos_pattern = ([^,]*)
gecos_name = $1
```

gecos_pattern	Use: *main*	Type: *string*	Default: *unset*

See **gecos_name** above.

gnutls_require_kx	Use: *main*	Type: *string*	Default: *unset*

This option controls the key exchange mechanisms when GnuTLS is used in an Exim server. For details, see section 39.5.

gnutls_require_mac	Use: *main*	Type: *string*	Default: *unset*

This option controls the MAC algorithms when GnuTLS is used in an Exim server. For details, see section 39.5.

gnutls_require_protocols	Use: *main*	Type: *string*	Default: *unset*

This option controls the protocols when GnuTLS is used in an Exim server. For details, see section 39.5.

headers_charset	Use: *main*	Type: *string*	Default: *see below*

This option sets a default character set for translating from encoded MIME "words" in header lines, when referenced by an *$h_xxx* expansion item. The default is the value of HEADERS_CHARSET in *Local/Makefile*. The ultimate default is ISO-8859-1. For more details see the description of header insertions in section 11.5.

header_maxsize	Use: *main*	Type: *integer*	Default: *see below*

This option controls the overall maximum size of a message's header section. The default is the value of HEADER_MAXSIZE in *Local/Makefile*; the default for that is 1M. Messages with larger header sections are rejected.

header_line_maxsize	Use: *main*	Type: *integer*	Default: *0*

This option limits the length of any individual header line in a message, after all the continuations have been joined together. Messages with individual header lines that are longer than the limit are rejected. The default value of zero means "no limit".

helo_accept_junk_hosts	Use: *main*	Type: *host list*†	Default: *unset*

Exim checks the syntax of HELO and EHLO commands for incoming SMTP mail, and gives an error response for invalid data. Unfortunately, there are some SMTP clients that send syntactic junk. They can be accommodated by setting this option. Note that this is a syntax check only. See **helo_verify_hosts** if you want to do semantic checking. See also **helo_allow_chars** for a way of extending the permitted character set.

helo_allow_chars	Use: *main*	Type: *string*	Default: *unset*

This option can be set to a string of rogue characters that are permitted in all EHLO and HELO names in addition to the standard letters, digits, hyphens, and dots. If you really must allow underscores, you can set

```
helo_allow_chars = _
```

Note that the value is one string, not a list.

helo_lookup_domains	Use: *main*	Type: *domain list*†	Default: `@:@[]`

If the domain given by a client in a HELO or EHLO command matches this list, a reverse lookup is done in order to establish the host's true name. The default forces a lookup if the client host gives the server's name or any of its IP addresses (in brackets), something that broken clients have been seen to do.

helo_try_verify_hosts	Use: *main*	Type: *host list*†	Default: *unset*

By default, Exim just checks the syntax of HELO and EHLO commands (see **helo_accept_junk_hosts** and **helo_allow_chars**). However, some sites like to do more extensive checking of the data supplied by these commands. The ACL condition `verify = helo` is provided to make this possible. Formerly, it was necessary also to set this option (**helo_try_verify_hosts**) to force the check to occur. From release 4.53 onwards, this is no longer necessary. If the check has not been done before `verify = helo` is encountered, it is done at that time. Consequently, this option is obsolete. Its specification is retained here for backwards compatibility.

When an EHLO or HELO command is received, if the calling host matches **helo_try_verify_hosts**, Exim checks that the host name given in the HELO or EHLO command either:

- is an IP literal matching the calling address of the host, or
- matches the host name that Exim obtains by doing a reverse lookup of the calling host address, or
- when looked up using *gethostbyname()* (or *getipnodebyname()* when available) yields the calling host address.

However, the EHLO or HELO command is not rejected if any of the checks fail. Processing continues, but the result of the check is remembered, and can be detected later in an ACL by the `verify = helo` condition.

helo_verify_hosts	Use: *main*	Type: *host list*†	Default: *unset*

Like **helo_try_verify_hosts**, this option is obsolete, and retained only for backwards compatibility. For hosts that match this option, Exim checks the host name given in the HELO or EHLO in the same way as for **helo_try_verify_hosts**. If the check fails, the HELO or EHLO command is rejected with a 550 error, and entries are written to the main and reject logs. If a MAIL command is received before EHLO or HELO, it is rejected with a 503 error.

hold_domains	Use: *main*	Type: *domain list*†	Default: *unset*

This option allows mail for particular domains to be held on the queue manually. The option is overridden if a message delivery is forced with the **-M**, **-qf**, **-Rf** or **-Sf** options, and also while testing or verifying addresses using **-bt** or **-bv**. Otherwise, if a domain matches an item in **hold_domains**, no routing or delivery for that address is done, and it is deferred every time the message is looked at.

This option is intended as a temporary operational measure for delaying the delivery of mail while some problem is being sorted out, or some new configuration tested. If you just want to delay the processing of some domains until a queue run occurs, you should use **queue_domains** or **queue_smtp_domains**, not **hold_domains**.

A setting of **hold_domains** does not override Exim's code for removing messages from the queue if they have been there longer than the longest retry time in any retry rule. If you want to hold messages for longer than the normal retry times, insert a dummy retry rule with a long retry time.

host_lookup	Use: *main*	Type: *host list*†	Default: *unset*

Exim does not look up the name of a calling host from its IP address unless it is required to compare against some host list, or the host matches **helo_try_verify_hosts** or **helo_verify_hosts**, or the host matches this option (which normally contains IP addresses rather than host names). The default configuration file contains

```
host_lookup = *
```

which causes a lookup to happen for all hosts. If the expense of these lookups is felt to be too great, the setting can be changed or removed.

After a successful reverse lookup, Exim does a forward lookup on the name it has obtained, to verify that it yields the IP address that it started with. If this check fails, Exim behaves as if the name lookup failed.

After any kind of failure, the host name (in *$sender_host_name*) remains unset, and *$host_lookup_failed* is set to the string "1". See also **dns_again_means_nonexist**, **helo_lookup_domains**, and `verify = reverse_host_lookup` in ACLs.

host_lookup_order	Use: *main*	Type: *string list*	Default: `bydns:byaddr`

This option specifies the order of different lookup methods when Exim is trying to find a host name from an IP address. The default is to do a DNS lookup first, and then to try a local lookup (using *gethostbyaddr()* or equivalent) if that fails. You can change the order of these lookups, or omit one entirely, if you want.

Warning: The "byaddr" method does not always yield aliases when there are multiple PTR records in the DNS and the IP address is not listed in */etc/hosts*. Different operating systems give different results in this case. That is why the default tries a DNS lookup first.

host_reject_connection	Use: *main*	Type: *host list*†	Default: *unset*

If this option is set, incoming SMTP calls from the hosts listed are rejected as soon as the connection is made. This option is obsolete, and retained only for backward compatibility, because nowadays the ACL specified by **acl_smtp_connect** can also reject incoming connections immediately.

The ability to give an immediate rejection (either by this option or using an ACL) is provided for use in unusual cases. Many hosts will just try again, sometimes without much delay. Normally, it is better to use an ACL to reject incoming messages at a later stage, such as after RCPT commands. See chapter 40.

hosts_connection_nolog	Use: *main*	Type: *host list*†	Default: *unset*

This option defines a list of hosts for which connection logging does not happen, even though the **smtp_connection** log selector is set. For example, you might want not to log SMTP connections from local processes, or from 127.0.0.1, or from your local LAN. This option is consulted in the main loop of the daemon; you should therefore strive to restrict its value to a short inline list of IP addresses and networks. To disable logging SMTP connections from local processes, you must create a host list with an empty item. For example:

```
hosts_connection_nolog = :
```

If the **smtp_connection** log selector is not set, this option has no effect.

hosts_treat_as_local	Use: *main*	Type: *domain list*†	Default: *unset*

If this option is set, any host names that match the domain list are treated as if they were the local host when Exim is scanning host lists obtained from MX records or other sources. Note that the value of this option is a domain list, not a host list, because it is always used to check host names, not IP addresses.

This option also applies when Exim is matching the special items `@mx_any`, `@mx_primary`, and `@mx_secondary` in a domain list (see section 10.8), and when checking the **hosts** option in the *smtp* transport for the local host (see the **allow_localhost** option in that transport). See also **local_interfaces**, **extra_local_interfaces**, and chapter 13, which contains a discussion about local network interfaces and recognizing the local host.

ibase_servers	Use: *main*	Type: *string list*	Default: *unset*

This option provides a list of InterBase servers and associated connection data, to be used in conjunction with *ibase* lookups (see section 9.21). The option is available only if Exim has been built with InterBase support.

ignore_bounce_errors_after	Use: *main*	Type: *time*	Default: *10w*

This option affects the processing of bounce messages that cannot be delivered, that is, those that suffer a permanent delivery failure. (Bounce messages that suffer temporary delivery failures are of course retried in the usual way.)

After a permanent delivery failure, bounce messages are frozen, because there is no sender to whom they can be returned. When a frozen bounce message has been on the queue for more than the given time, it is unfrozen at the next queue run, and a further delivery is attempted. If delivery fails again, the bounce message is discarded. This makes it possible to keep failed bounce messages around for a shorter time than the normal maximum retry time for frozen messages. For example,

```
ignore_bounce_errors_after = 12h
```

retries failed bounce message deliveries after 12 hours, discarding any further failures. If the value of this option is set to a zero time period, bounce failures are discarded immediately. Setting a very long time (as in the default value) has the effect of disabling this option. For ways of automatically dealing with other kinds of frozen message, see **auto_thaw** and **timeout_frozen_after**.

ignore_fromline_hosts	Use: *main*	Type: *host list*†	Default: *unset*

Some broken SMTP clients insist on sending a UUCP-like "From " line before the headers of a message. By default this is treated as the start of the message's body, which means that any following headers are not recognized as such. Exim can be made to ignore it by setting **ignore_fromline_hosts** to match those hosts that insist on sending it. If the sender is actually a local process rather than a remote host, and is using **-bs** to inject the messages, **ignore_fromline_local** must be set to achieve this effect.

ignore_fromline_local	Use: *main*	Type: *boolean*	Default: *false*

See **ignore_fromline_hosts** above.

keep_malformed	Use: *main*	Type: *time*	Default: *4d*

This option specifies the length of time to keep messages whose spool files have been corrupted in some way. This should, of course, never happen. At the next attempt to deliver such a message, it gets removed. The incident is logged.

ldap_default_servers	Use: *main*	Type: *string list*	Default: *unset*

This option provides a list of LDAP servers which are tried in turn when an LDAP query does not contain a server. See section 9.14 for details of LDAP queries. This option is available only when Exim has been built with LDAP support.

ldap_version	Use: *main*	Type: *integer*	Default: *unset*

This option can be used to force Exim to set a specific protocol version for LDAP. If it option is unset, it is shown by the **-bP** command line option as -1. When this is the case, the default is 3 if LDAP_VERSION3 is defined in the LDAP headers; otherwise it is 2. This option is available only when Exim has been built with LDAP support.

local_from_check	Use: *main*	Type: *boolean*	Default: *true*

When a message is submitted locally (that is, not over a TCP/IP connection) by an untrusted user, Exim removes any existing *Sender:* header line, and checks that the *From:* header line matches the login of the calling user and the domain specified by **qualify_domain**.

Note: An unqualified address (no domain) in the *From:* header in a locally submitted message is automatically qualified by Exim, unless the **-bnq** command line option is used.

You can use **local_from_prefix** and **local_from_suffix** to permit affixes on the local part. If the *From:* header line does not match, Exim adds a *Sender:* header with an address constructed from the calling user's login and the default qualify domain.

If **local_from_check** is set false, the *From:* header check is disabled, and no *Sender:* header is ever added. If, in addition, you want to retain *Sender:* header lines supplied by untrusted users, you must also set **local_sender_retain** to be true.

These options affect only the header lines in the message. The envelope sender is still forced to be the login id at the qualify domain unless **untrusted_set_sender** permits the user to supply an envelope sender.

For messages received over TCP/IP, an ACL can specify "submission mode" to request similar header line checking. See section 44.16, which has more details about *Sender:* processing.

local_from_prefix	Use: *main*	Type: *string*	Default: *unset*

When Exim checks the *From:* header line of locally submitted messages for matching the login id (see **local_from_check** above), it can be configured to ignore certain prefixes and suffixes in the local part of the address. This is done by setting **local_from_prefix** and/or **local_from_suffix** to appropriate lists, in the same form as the **local_part_prefix** and **local_part_suffix** router options (see chapter 15). For example, if

```
local_from_prefix = *-
```

is set, a *From:* line containing

```
From: anything-user@your.domain.example
```

will not cause a *Sender:* header to be added if *user@your.domain.example* matches the actual sender address that is constructed from the login name and qualify domain.

local_from_suffix	Use: *main*	Type: *string*	Default: *unset*

See **local_from_prefix** above.

local_interfaces	Use: *main*	Type: *string list*	Default: *see below*

This option controls which network interfaces are used by the daemon for listening; they are also used to identify the local host when routing. Chapter 13 contains a full description of this option and the related options **daemon_smtp_ports**, **extra_local_interfaces**, **hosts_treat_as_local**, and **tls_on_connect_ports**. The default value for **local_interfaces** is

```
local_interfaces = 0.0.0.0
```

when Exim is built without IPv6 support; otherwise it is

```
local_interfaces = <; ::0 ; 0.0.0.0
```

local_scan_timeout	Use: *main*	Type: *time*	Default: *5m*

This timeout applies to the *local_scan()* function (see chapter 42). Zero means "no timeout". If the timeout is exceeded, the incoming message is rejected with a temporary error if it is an SMTP message. For a non-SMTP message, the message is dropped and Exim ends with a non-zero code. The incident is logged on the main and reject logs.

local_sender_retain	Use: *main*	Type: *boolean*	Default: *false*

When a message is submitted locally (that is, not over a TCP/IP connection) by an untrusted user, Exim removes any existing *Sender:* header line. If you do not want this to happen, you must set **local_sender_retain**, and you must also set **local_from_check** to be false (Exim will complain if you do not). See also the ACL modifier `control = suppress_local_fixups`. Section 44.16 has more details about *Sender:* processing.

localhost_number	Use: *main*	Type: *string*†	Default: *unset*

Exim's message ids are normally unique only within the local host. If uniqueness among a set of hosts is required, each host must set a different value for the **localhost_number** option. The string is expanded immediately after reading the configuration file (so that a number can be computed from the host name, for example) and the result of the expansion must be a number in the range 0–16 (or 0–10 on operating systems with case-insensitive file systems). This is available in subsequent string expansions via the variable *$localhost_number*. When **localhost_number is set**, the final two characters of the message id, instead of just being a fractional part of the time, are computed from the time and the local host number as described in section 3.4.

log_file_path	Use: *main*	Type: *string list*†	Default: *set at compile time*

This option sets the path which is used to determine the names of Exim's log files, or indicates that logging is to be to syslog, or both. It is expanded when Exim is entered, so it can, for example, contain a reference to the host name. If no specific path is set for the log files at compile or run time, they are written in a sub-directory called *log* in Exim's spool directory. Chapter 49 contains further details about Exim's logging, and section 49.1 describes how the contents of **log_file_path** are used. If this string is fixed at your installation (contains no expansion variables) it is recommended that you do not set this option in the configuration file, but instead supply the path using LOG_FILE_PATH in *Local/Makefile* so that it is available to Exim for logging errors detected early on – in particular, failure to read the configuration file.

log_selector	Use: *main*	Type: *string*	Default: *unset*

This option can be used to reduce or increase the number of things that Exim writes to its log files. Its argument is made up of names preceded by plus or minus characters. For example:

```
log_selector = +arguments -retry_defer
```

A list of possible names and what they control is given in the chapter on logging, in section 49.15.

log_timezone	Use: *main*	Type: *boolean*	Default: *false*

By default, the timestamps on log lines are in local time without the timezone. This means that if your timezone changes twice a year, the timestamps in log lines are ambiguous for an hour when the clocks go back. One way of avoiding this problem is to set the timezone to UTC. An alternative is to set **log_timezone** true. This turns on the addition of the timezone offset to timestamps in log lines. Turning on this option can add quite a lot to the size of log files because each line is extended by 6 characters. Note that the *$tod_log* variable contains the log timestamp without the zone, but there is another variable called *$tod_zone* that contains just the timezone offset.

lookup_open_max	Use: *main*	Type: *integer*	Default: *25*

This option limits the number of simultaneously open files for single-key lookups that use regular files (that is, *lsearch*, *dbm*, and *cdb*). Exim normally keeps these files open during routing, because often the same file is required several times. If the limit is reached, Exim closes the least recently used file. Note that if you are using the *ndbm* library, it actually opens two files for each logical DBM database, though it still counts as one for the purposes of **lookup_open_max**. If you are getting "too many open files" errors with NDBM, you need to reduce the value of **lookup_open_max**.

max_username_length	Use: *main*	Type: *integer*	Default: *0*

Some operating systems are broken in that they truncate long arguments to *getpwnam()* to eight characters, instead of returning "no such user". If this option is set greater than zero, any attempt to call *getpwnam()* with an argument that is longer behaves as if *getpwnam()* failed.

message_body_newlines	Use: *main*	Type: *bool*	Default: *false*

By default, newlines in the message body are replaced by spaces when setting the *$message_body* and *$message_body_end* expansion variables. If this option is set true, this no longer happens.

message_body_visible	Use: *main*	Type: *integer*	Default: *500*

This option specifies how much of a message's body is to be included in the *$message_body* and *$message_body_end* expansion variables.

message_id_header_domain	Use: *main*	Type: *string*†	Default: *unset*

If this option is set, the string is expanded and used as the right hand side (domain) of the *Message-ID:* header that Exim creates if a locally-originated incoming message does not have one. "Locally-originated" means "not received over TCP/IP." Otherwise, the primary host name is used. Only letters, digits, dot and hyphen are accepted; any other characters are replaced by hyphens. If the expansion is forced to fail, or if the result is an empty string, the option is ignored.

message_id_header_text	Use: *main*	Type: *string*†	Default: *unset*

If this variable is set, the string is expanded and used to augment the text of the *Message-id:* header that Exim creates if a locally-originated incoming message does not have one. The text of this header is required by RFC 2822 to take the form of an address. By default, Exim uses its internal message id as the local part, and the primary host name as the domain. If this option is set, it is expanded, and provided the expansion is not forced to fail, and does not yield an empty string, the result is inserted into the header immediately before the @, separated from the internal message id by a dot. Any characters that are illegal in an address are automatically converted into hyphens. This means that variables such as *$tod_log* can be used, because the spaces and colons will become hyphens.

message_logs	Use: *main*	Type: *boolean*	Default: *true*

If this option is turned off, per-message log files are not created in the *msglog* spool sub-directory. This reduces the amount of disk I/O required by Exim, by reducing the number of files involved in handling a message from a minimum of four (header spool file, body spool file, delivery journal, and per-message log) to three. The other major I/O activity is Exim's main log, which is not affected by this option.

message_size_limit	Use: *main*	Type: *string*†	Default: *50M*

This option limits the maximum size of message that Exim will process. The value is expanded for each incoming connection so, for example, it can be made to depend on the IP address of the remote host for messages arriving via TCP/IP. After expansion, the value must be a sequence of decimal digits, optionally followed by K or M.

Note: This limit cannot be made to depend on a message's sender or any other properties of an individual message, because it has to be advertised in the server's response to EHLO. String expansion failure causes a temporary error. A value of zero means no limit, but its use is not recommended. See also **bounce_return_size_limit**.

Incoming SMTP messages are failed with a 552 error if the limit is exceeded; locally-generated messages either get a stderr message or a delivery failure message to the sender, depending on the **-oe** setting. Rejection of an oversized message is logged in both the main and the reject logs. See also the generic transport option **message_size_limit**, which limits the size of message that an individual transport can process.

move_frozen_messages	Use: *main*	Type: *boolean*	Default: *false*

This option, which is available only if Exim has been built with the setting

```
SUPPORT_MOVE_FROZEN_MESSAGES=yes
```

in *Local/Makefile*, causes frozen messages and their message logs to be moved from the *input* and *msglog* directories on the spool to *Finput* and *Fmsglog*, respectively. There is currently no support in Exim or the standard utilities for handling such moved messages, and they do not show up in lists generated by **-bp** or by the Exim monitor.

mua_wrapper	Use: *main*	Type: *boolean*	Default: *false*

Setting this option true causes Exim to run in a very restrictive mode in which it passes messages synchronously to a smart host. Chapter 48 contains a full description of this facility.

mysql_servers	Use: *main*	Type: *string list*	Default: *unset*

This option provides a list of MySQL servers and associated connection data, to be used in conjunction with *mysql* lookups (see section 9.21). The option is available only if Exim has been built with MySQL support.

never_users	Use: *main*	Type: *string list*†	Default: *unset*

This option is expanded just once, at the start of Exim's processing. Local message deliveries are normally run in processes that are setuid to the recipient, and remote deliveries are normally run under Exim's own uid and gid. It is usually desirable to prevent any deliveries from running as root, as a safety precaution.

When Exim is built, an option called FIXED_NEVER_USERS can be set to a list of users that must not be used for local deliveries. This list is fixed in the binary and cannot be overridden by the configuration file. By default, it contains just the single user name "root". The **never_users** runtime option can be used to add more users to the fixed list.

If a message is to be delivered as one of the users on the fixed list or the **never_users** list, an error occurs, and delivery is deferred. A common example is

```
never_users = root:daemon:bin
```

Including root is redundant if it is also on the fixed list, but it does no harm. This option overrides the **pipe_as_creator** option of the *pipe* transport driver.

oracle_servers	Use: *main*	Type: *string list*	Default: *unset*

This option provides a list of Oracle servers and associated connection data, to be used in conjunction with *oracle* lookups (see section 9.21). The option is available only if Exim has been built with Oracle support.

percent_hack_domains	Use: *main*	Type: *domain list*†	Default: *unset*

The "percent hack" is the convention whereby a local part containing a percent sign is re-interpreted as a new email address, with the percent replaced by @. This is sometimes called "source routing", though that term is also applied to RFC 2822 addresses that begin with an @ character. If this option is set, Exim implements the percent facility for those domains listed, but no others. This happens before an incoming SMTP address is tested against an ACL.

Warning: The "percent hack" has often been abused by people who are trying to get round relaying restrictions. For this reason, it is best avoided if at all possible. Unfortunately, a number of less security-conscious MTAs implement it unconditionally. If you are running Exim on a gateway host, and routing mail through to internal MTAs without processing the local parts, it is a good idea to reject recipient addresses with percent characters in their local parts. Exim's default configuration does this.

perl_at_start	Use: *main*	Type: *boolean*	Default: *false*

This option is available only when Exim is built with an embedded Perl interpreter. See chapter 12 for details of its use.

perl_startup	Use: *main*	Type: *string*	Default: *unset*

This option is available only when Exim is built with an embedded Perl interpreter. See chapter 12 for details of its use.

pgsql_servers	Use: *main*	Type: *string list*	Default: *unset*

This option provides a list of PostgreSQL servers and associated connection data, to be used in conjunction with *pgsql* lookups (see section 9.21). The option is available only if Exim has been built with PostgreSQL support.

pid_file_path	Use: *main*	Type: *string*†	Default: *set at compile time*

This option sets the name of the file to which the Exim daemon writes its process id. The string is expanded, so it can contain, for example, references to the host name:

```
pid_file_path = /var/log/$primary_hostname/exim.pid
```

If no path is set, the pid is written to the file *exim-daemon.pid* in Exim's spool directory. The value set by the option can be overridden by the **-oP** command line option. A pid file is not written if a "non-standard" daemon is run by means of the **-oX** option, unless a path is explicitly supplied by **-oP**.

pipelining_advertise_hosts	Use: *main*	Type: *host list*†	Default: *

This option can be used to suppress the advertisement of the SMTP PIPELINING extension to specific hosts. See also the **no_pipelining** control in section 40.20. When PIPELINING is not advertised and **smtp_enforce_sync** is true, an Exim server enforces strict synchronization for each SMTP command and response. When PIPELINING is advertised, Exim assumes that clients will use it; "out of order" commands that are "expected" do not count as protocol errors (see **smtp_max_synprot_errors**).

preserve_message_logs	Use: *main*	Type: *boolean*	Default: *false*

If this option is set, message log files are not deleted when messages are completed. Instead, they are moved to a sub-directory of the spool directory called *msglog.OLD*, where they remain available for statistical or debugging purposes. This is a dangerous option to set on systems with any appreciable volume of mail. Use with care!

primary_hostname	Use: *main*	Type: *string*	Default: *see below*

This specifies the name of the current host. It is used in the default EHLO or HELO command for outgoing SMTP messages (changeable via the **helo_data** option in the *smtp* transport), and as the default for **qualify_domain**. The value is also used by default in some SMTP response messages from an Exim server. This can be changed dynamically by setting **smtp_active_hostname**.

If **primary_hostname** is not set, Exim calls *uname()* to find the host name. If this fails, Exim panics and dies. If the name returned by *uname()* contains only one component, Exim passes it to *gethostbyname()* (or *getipnodebyname()* when available) in order to obtain the fully qualified version. The variable *$primary_hostname* contains the host name, whether set explicitly by this option, or defaulted.

print_topbitchars	Use: *main*	Type: *boolean*	Default: *false*

By default, Exim considers only those characters whose codes lie in the range 32–126 to be printing characters. In a number of circumstances (for example, when writing log entries) non-printing characters are converted into escape sequences, primarily to avoid messing up the layout. If **print_topbitchars** is set, code values of 128 and above are also considered to be printing characters.

This option also affects the header syntax checks performed by the *autoreply* transport, and whether Exim uses RFC 2047 encoding of the user's full name when constructing From: and Sender: addresses (as described in section 44.18). Setting this option can cause Exim to generate eight bit message headers that do not conform to the standards.

process_log_path	Use: *main*	Type: *string*	Default: *unset*

This option sets the name of the file to which an Exim process writes its "process log" when sent a USR1 signal. This is used by the *exiwhat* utility script. If this option is unset, the file called *exim-process.info* in Exim's spool directory is used. The ability to specify the name explicitly can be useful in environments where two different Exims are running, using different spool directories.

prod_requires_admin	Use: *main*	Type: *boolean*	Default: *true*

The **-M**, **-R**, and **-q** command-line options require the caller to be an admin user unless **prod_requires_admin** is set false. See also **queue_list_requires_admin**.

qualify_domain	Use: *main*	Type: *string*	Default: *see below*

This option specifies the domain name that is added to any envelope sender addresses that do not have a domain qualification. It also applies to recipient addresses if **qualify_recipient** is not set. Unqualified addresses are accepted by default only for locally-generated messages. Qualification is also applied to addresses in header lines such as *From:* and *To:* for locally-generated messages, unless the **-bnq** command line option is used.

Messages from external sources must always contain fully qualified addresses, unless the sending host matches **sender_unqualified_hosts** or **recipient_unqualified_hosts** (as appropriate), in which case incoming addresses are qualified with **qualify_domain** or **qualify_recipient** as necessary. Internally,

Exim always works with fully qualified envelope addresses. If **qualify_domain** is not set, it defaults to the **primary_hostname** value.

qualify_recipient	Use: *main*	Type: *string*	Default: *see below*

This option allows you to specify a different domain for qualifying recipient addresses to the one that is used for senders. See **qualify_domain** above.

queue_domains	Use: *main*	Type: *domain list†*	Default: *unset*

This option lists domains for which immediate delivery is not required. A delivery process is started whenever a message is received, but only those domains that do not match are processed. All other deliveries wait until the next queue run. See also **hold_domains** and **queue_smtp_domains**.

queue_list_requires_admin	Use: *main*	Type: *boolean*	Default: *true*

The **-bp** command-line option, which lists the messages that are on the queue, requires the caller to be an admin user unless **queue_list_requires_admin** is set false. See also **prod_requires_admin**.

queue_only	Use: *main*	Type: *boolean*	Default: *false*

If **queue_only** is set, a delivery process is not automatically started whenever a message is received. Instead, the message waits on the queue for the next queue run. Even if **queue_only** is false, incoming messages may not get delivered immediately when certain conditions (such as heavy load) occur.

The **-odq** command line has the same effect as **queue_only**. The **-odb** and **-odi** command line options override **queue_only** unless **queue_only_override** is set false. See also **queue_only_file**, **queue_only_load**, and **smtp_accept_queue**.

queue_only_file	Use: *main*	Type: *string*	Default: *unset*

This option can be set to a colon-separated list of absolute path names, each one optionally preceded by "smtp". When Exim is receiving a message, it tests for the existence of each listed path using a call to *stat()*. For each path that exists, the corresponding queueing option is set. For paths with no prefix, **queue_only** is set; for paths prefixed by "smtp", **queue_smtp_domains** is set to match all domains. So, for example,

```
queue_only_file = smtp/some/file
```

causes Exim to behave as if **queue_smtp_domains** were set to "*" whenever */some/file* exists.

queue_only_load	Use: *main*	Type: *fixed-point*	Default: *unset*

If the system load average is higher than this value, incoming messages from all sources are queued, and no automatic deliveries are started. If this happens during local or remote SMTP input, all subsequent messages received on the same SMTP connection are queued by default, whatever happens to the load in the meantime, but this can be changed by setting **queue_only_load_latch** false.

Deliveries will subsequently be performed by queue runner processes. This option has no effect on ancient operating systems on which Exim cannot determine the load average. See also **deliver_queue_load_max** and **smtp_load_reserve**.

queue_only_load_latch	Use: *main*	Type: *boolean*	Default: *true*

When this option is true (the default), once one message has been queued because the load average is higher than the value set by **queue_only_load**, all subsequent messages received on the same SMTP connection are also queued. This is a deliberate choice; even though the load average may fall below the threshold, it doesn't seem right to deliver later messages on the same connection when not delivering earlier ones. However, there are special circumstances such as very long-lived connections from scanning appliances where this is not the best strategy. In such cases, **queue_only_load_latch** should be set false. This causes the value of the load average to be re-evaluated for each message.

queue_only_override	Use: *main*	Type: *boolean*	Default: *true*

When this option is true, the **-od***x* command line options override the setting of **queue_only** or **queue_only_file** in the configuration file. If **queue_only_override** is set false, the **-od***x* options cannot be used to override; they are accepted, but ignored.

queue_run_in_order	Use: *main*	Type: *boolean*	Default: *false*

If this option is set, queue runs happen in order of message arrival instead of in an arbitrary order. For this to happen, a complete list of the entire queue must be set up before the deliveries start. When the queue is all held in a single directory (the default), a single list is created for both the ordered and the non-ordered cases. However, if **split_spool_directory** is set, a single list is not created when **queue_run_in_order** is false. In this case, the sub-directories are processed one at a time (in a random order), and this avoids setting up one huge list for the whole queue. Thus, setting **queue_run_in_order** with **split_spool_directory** may degrade performance when the queue is large, because of the extra work in setting up the single, large list. In most situations, **queue_run_in_order** should not be set.

queue_run_max	Use: *main*	Type: *integer*	Default: *5*

This controls the maximum number of queue runner processes that an Exim daemon can 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 another one. This can happen with very large queues and/or very sluggish deliveries. This option does not, however, interlock with other processes, so additional queue runners can be started by other means, or by killing and restarting the daemon.

Setting this option to zero does not suppress queue runs; rather, it disables the limit, allowing any number of simultaneous queue runner processes to be run. If you do not want queue runs to occur, omit the **-q***xx* setting on the daemon's command line.

queue_smtp_domains	Use: *main*	Type: *domain list*†	Default: *unset*

When this option is set, a delivery process is started whenever a message is received, routing is performed, and local deliveries take place. However, if any SMTP deliveries are required for domains that match **queue_smtp_domains**, they are not immediately delivered, but instead the message waits on the queue for the next queue run. Since routing of the message has taken place, Exim knows to which remote hosts it must be delivered, and so when the queue run happens, multiple messages for the same host are delivered over a single SMTP connection. The **-odqs** command line option causes all SMTP deliveries to be queued in this way, and is equivalent to setting **queue_smtp_domains** to "*". See also **hold_domains** and **queue_domains**.

receive_timeout	Use: *main*	Type: *time*	Default: *0s*

This option sets the timeout for accepting a non-SMTP message, that is, the maximum time that Exim waits when reading a message on the standard input. If the value is zero, it will wait for ever. This setting is overridden by the **-or** command line option. The timeout for incoming SMTP messages is controlled by **smtp_receive_timeout**.

received_header_text	Use: *main*	Type: *string*†	Default: *see below*

This string defines the contents of the *Received:* message header that is added to each message, except for the timestamp, which is automatically added on at the end (preceded by a semicolon). The string is expanded each time it is used. If the expansion yields an empty string, no *Received:* header line is added to the message. Otherwise, the string should start with the text "Received:" and conform to the RFC 2822 specification for *Received:* header lines. The default setting is:

```
received_header_text = Received: \
  ${if def:sender_rcvhost {from $sender_rcvhost\n\t}\
  {${if def:sender_ident \
  {from ${quote_local_part:$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)\n\t\
  ${if def:sender_address \
  {(envelope-from <$sender_address>)\n\t}}\
  id $message_exim_id\
  ${if def:received_for {\n\tfor $received_for}}
```

The reference to the TLS cipher is omitted when Exim is built without TLS support. The use of conditional expansions ensures that this works for both locally generated messages and messages received from remote hosts, giving header lines such as the following:

```
Received: from scrooge.carol.example ([192.168.12.25] ident=root)
by marley.carol.example with esmtp (Exim 4.00)
(envelope-from <bob@carol.example>)
id 16IOWa-00019l-00
for chas@dickens.example; Tue, 25 Dec 2001 14:43:44 +0000
Received: by scrooge.carol.example with local (Exim 4.00)
id 16IOWW-000083-00; Tue, 25 Dec 2001 14:43:41 +0000
```

Until the body of the message has been received, the timestamp is the time when the message started to be received. Once the body has arrived, and all policy checks have taken place, the timestamp is updated to the time at which the message was accepted.

received_headers_max	Use: *main*	Type: *integer*	Default: *30*

When a message is to be delivered, the number of *Received:* headers is counted, and 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.

recipient_unqualified_hosts	Use: *main*	Type: *host list*†	Default: *unset*

This option lists those hosts from which Exim is prepared to accept unqualified recipient addresses in message envelopes. The addresses are made fully qualified by the addition of the **qualify_recipient** value. This option also affects message header lines. Exim does not reject unqualified recipient addresses in headers, but it qualifies them only if the message came from a host that matches

recipient_unqualified_hosts, or if the message was submitted locally (not using TCP/IP), and the **-bnq** option was not set.

recipients_max	Use: *main*	Type: *integer*	Default: *0*

If this option is set greater than zero, it specifies the maximum number of original recipients for any message. Additional recipients that are generated by aliasing or forwarding do not count. SMTP messages get a 452 response for all recipients over the limit; earlier recipients are delivered as normal. Non-SMTP messages with too many recipients are failed, and no deliveries are done.

Note: The RFCs specify that an SMTP server should accept at least 100 RCPT commands in a single message.

recipients_max_reject	Use: *main*	Type: *boolean*	Default: *false*

If this option is set true, Exim rejects SMTP messages containing too many recipients by giving 552 errors to the surplus RCPT commands, and a 554 error to the eventual DATA command. Otherwise (the default) it gives a 452 error to the surplus RCPT commands and accepts the message on behalf of the initial set of recipients. The remote server should then re-send the message for the remaining recipients at a later time.

remote_max_parallel	Use: *main*	Type: *integer*	Default: *2*

This option controls parallel delivery of one message to a number of remote hosts. If the value is less than 2, parallel delivery is disabled, and Exim does all the remote deliveries for a message one by one. Otherwise, if a single message has to be delivered to more than one remote host, or if several copies have to be sent to the same remote host, up to **remote_max_parallel** deliveries are done simultaneously. If more than **remote_max_parallel** deliveries are required, the maximum number of processes are started, and as each one finishes, another is begun. The order of starting processes is the same as if sequential delivery were being done, and can be controlled by the **remote_sort_domains** option. If parallel delivery takes place while running with debugging turned on, the debugging output from each delivery process is tagged with its process id.

This option controls only the maximum number of parallel deliveries for one message in one Exim delivery process. Because Exim has no central queue manager, there is no way of controlling the total number of simultaneous deliveries if the configuration allows a delivery attempt as soon as a message is received.

If you want to control the total number of deliveries on the system, you need to set the **queue_only** option. This ensures that all incoming messages are added to the queue without starting a delivery process. Then set up an Exim daemon to start queue runner processes at appropriate intervals (probably fairly often, for example, every minute), and limit the total number of queue runners by setting the **queue_run_max** parameter. Because each queue runner delivers only one message at a time, the maximum number of deliveries that can then take place at once is **queue_run_max** multiplied by **remote_max_parallel**.

If it is purely remote deliveries you want to control, use **queue_smtp_domains** instead of **queue_only**. This has the added benefit of doing the SMTP routing before queueing, so that several messages for the same host will eventually get delivered down the same connection.

remote_sort_domains	Use: *main*	Type: *domain list*†	Default: *unset*

When there are a number of remote deliveries for a message, they are sorted by domain into the order given by this list. For example,

```
remote_sort_domains = *.cam.ac.uk:*.uk
```

would attempt to deliver to all addresses in the *cam.ac.uk* domain first, then to those in the **uk** domain, then to any others.

retry_data_expire	Use: *main*	Type: *time*	Default: *7d*

This option sets a "use before" time on retry information in Exim's hints database. Any older retry data is ignored. This means that, for example, once a host has not been tried for 7 days, Exim behaves as if it has no knowledge of past failures.

retry_interval_max	Use: *main*	Type: *time*	Default: *24h*

Chapter 32 describes Exim's mechanisms for controlling the intervals between delivery attempts for messages that cannot be delivered straight away. This option sets an overall limit to the length of time between retries. It cannot be set greater than 24 hours; any attempt to do so forces the default value.

return_path_remove	Use: *main*	Type: *boolean*	Default: *true*

RFC 2821, section 4.4, states that an SMTP server must insert a *Return-path:* header line into a message when it makes a "final delivery". The *Return-path:* header preserves the sender address as received in the MAIL command. This description implies that this header should not be present in an incoming message. If **return_path_remove** is true, any existing *Return-path:* headers are removed from messages at the time they are received. Exim's transports have options for adding *Return-path:* headers at the time of delivery. They are normally used only for final local deliveries.

return_size_limit	Use: *main*	Type: *integer*	Default: *100K*

This option is an obsolete synonym for **bounce_return_size_limit**.

rfc1413_hosts	Use: *main*	Type: *host list*†	Default: *

RFC 1413 identification calls are made to any client host which matches an item in the list.

rfc1413_query_timeout	Use: *main*	Type: *time*	Default: *5s*

This sets the timeout on RFC 1413 identification calls. If it is set to zero, no RFC 1413 calls are ever made.

sender_unqualified_hosts	Use: *main*	Type: *host list*†	Default: *unset*

This option lists those hosts from which Exim is prepared to accept unqualified sender addresses. The addresses are made fully qualified by the addition of **qualify_domain**. This option also affects message header lines. Exim does not reject unqualified addresses in headers that contain sender addresses, but it qualifies them only if the message came from a host that matches **sender_unqualified_hosts**, or if the message was submitted locally (not using TCP/IP), and the **-bnq** option was not set.

smtp_accept_keepalive	Use: *main*	Type: *boolean*	Default: *true*

This option controls the setting of the SO_KEEPALIVE option on incoming TCP/IP socket connections. When set, it causes the kernel to probe idle connections periodically, by sending packets with "old" sequence numbers. The other end of the connection should send an acknowledgment if the connection is still okay or a reset if the connection has been aborted. The reason for doing this is that it has the beneficial effect of freeing up certain types of connection that can get stuck when the remote

host is disconnected without tidying up the TCP/IP call properly. The keepalive mechanism takes several hours to detect unreachable hosts.

smtp_accept_max	Use: *main*	Type: *integer*	Default: *20*

This option 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, no limit is applied. However, it is required to be non-zero if either **smtp_accept_max_per_host** or **smtp_accept_queue** is set. See also **smtp_accept_reserve** and **smtp_load_reserve**.

A new SMTP connection is immediately rejected if the **smtp_accept_max** limit has been reached. If not, Exim first checks **smtp_accept_max_per_host**. If that limit has not been reached for the client host, **smtp_accept_reserve** and **smtp_load_reserve** are then checked before accepting the connection.

smtp_accept_max_nonmail	Use: *main*	Type: *integer*	Default: *10*

Exim counts the number of "non-mail" commands in an SMTP session, and drops the connection if there are too many. This option defines "too many". The check catches some denial-of-service attacks, repeated failing AUTHs, or a mad client looping sending EHLO, for example. The check is applied only if the client host matches **smtp_accept_max_nonmail_hosts**.

When a new message is expected, one occurrence of RSET is not counted. This allows a client to send one RSET between messages (this is not necessary, but some clients do it). Exim also allows one uncounted occurrence of HELO or EHLO, and one occurrence of STARTTLS between messages. After starting up a TLS session, another EHLO is expected, and so it too is not counted. The first occurrence of AUTH in a connection, or immediately following STARTTLS is not counted. Otherwise, all commands other than MAIL, RCPT, DATA, and QUIT are counted.

smtp_accept_max_nonmail_hosts	Use: *main*	Type: *host list*†	Default: *

You can control which hosts are subject to the **smtp_accept_max_nonmail** check by setting this option. The default value makes it apply to all hosts. By changing the value, you can exclude any badly-behaved hosts that you have to live with.

smtp_accept_max_per_ connection	Use: *main*	Type: *integer*	Default: *1000*

The value of this option limits the number of MAIL commands that Exim is prepared to accept over a single SMTP connection, whether or not each command results in the transfer of a message. After the limit is reached, a 421 response is given to subsequent MAIL commands. This limit is a safety precaution against a client that goes mad (incidents of this type have been seen).

smtp_accept_max_per_host	Use: *main*	Type: *string*†	Default: *unset*

This option restricts the number of simultaneous IP connections from a single host (strictly, from a single IP address) to the Exim daemon. The option is expanded, to enable different limits to be applied to different hosts by reference to *$sender_host_address*. Once the limit is reached, additional connection attempts from the same host are rejected with error code 421. This is entirely independent of **smtp_accept_reserve**. The option's default value of zero imposes no limit. If this option is set greater than zero, it is required that **smtp_accept_max** be non-zero.

Warning: When setting this option you should not use any expansion constructions that take an appreciable amount of time. The expansion and test happen in the main daemon loop, in order to reject additional connections without forking additional processes (otherwise a denial-of-service

attack could cause a vast number or processes to be created). While the daemon is doing this processing, it cannot accept any other incoming connections.

smtp_accept_queue	Use: *main*	Type: *integer*	Default: *0*

If the number of simultaneous incoming SMTP connections being handled via the listening daemon exceeds this value, messages received by SMTP are just placed on the queue; no delivery processes are started automatically. The count is fixed at the start of an SMTP connection. It cannot be updated in the subprocess that receives messages, and so the queueing or not queueing applies to all messages received in the same connection.

A value of zero implies no limit, and clearly any non-zero 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***x* command line options.

smtp_accept_queue_per_connection	Use: *main*	Type: *integer*	Default: *10*

This option limits the number of delivery processes that Exim starts automatically when receiving messages via SMTP, whether via the daemon or by the use of **-bs** or **-bS**. If the value of the option is greater than zero, and the number of messages received in a single SMTP session exceeds this number, subsequent messages are placed on the queue, but no delivery processes are started. This helps to limit the number of Exim processes when a server restarts after downtime and there is a lot of mail waiting for it on other systems. On large systems, the default should probably be increased, and on dial-in client systems it should probably be set to zero (that is, disabled).

smtp_accept_reserve	Use: *main*	Type: *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**. The value set in **smtp_accept_max** includes this reserve pool. The specified hosts are not restricted to this number of connections; the option specifies a minimum number of connection slots for them, not a maximum. It is a guarantee that this group of hosts can always get at least **smtp_accept_reserve** connections. However, the limit specified by **smtp_accept_max_per_host** is still applied to each individual host.

For example, if **smtp_accept_max** is set to 50 and **smtp_accept_reserve** is set to 5, once there are 45 active connections (from any hosts), new connections are accepted only from hosts listed in **smtp_reserve_hosts**, provided the other criteria for acceptance are met.

smtp_active_hostname	Use: *main*	Type: *string*†	Default: *unset*

This option is provided for multi-homed servers that want to masquerade as several different hosts. At the start of an incoming SMTP connection, its value is expanded and used instead of the value of *$primary_hostname* in SMTP responses. For example, it is used as domain name in the response to an incoming HELO or EHLO command.

The active hostname is placed in the *$smtp_active_hostname* variable, which is saved with any messages that are received. It is therefore available for use in routers and transports when the message is later delivered.

If this option is unset, or if its expansion is forced to fail, or if the expansion results in an empty string, the value of *$primary_hostname* is used. Other expansion failures cause a message to be written to the main and panic logs, and the SMTP command receives a temporary error. Typically, the value of **smtp_active_hostname** depends on the incoming interface address. For example:

```
smtp_active_hostname = ${if eq{$received_ip_address}{10.0.0.1}\
  {cox.mydomain}{box.mydomain}}
```

Although *$smtp_active_hostname* is primarily concerned with incoming messages, it is also used as the default for HELO commands in callout verification if there is no remote transport from which to obtain a **helo_data** value.

smtp_banner	Use: *main*	Type: *string†*	Default: *see below*

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 = $smtp_active_hostname ESMTP Exim \
  $version_number $tod_full
```

Failure to expand the string causes a panic error. If you want to create a multiline response to the initial SMTP connection, 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_check_spool_space	Use: *main*	Type: *boolean*	Default: *true*

When this option is set, if an incoming SMTP session encounters the SIZE option on a MAIL command, it checks that there is enough space in the spool directory's partition to accept a message of that size, while still leaving free the amount specified by **check_spool_space** (even if that value is zero). If there isn't enough space, a temporary error code is returned.

smtp_connect_backlog	Use: *main*	Type: *integer*	Default: *20*

This option 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. At least, that is what the manuals say; in some circumstances such connection attempts have been observed to time out instead. For large systems it is probably a good idea to increase the value (to 50, say). It also gives some protection against denial-of-service attacks by SYN flooding.

smtp_enforce_sync	Use: *main*	Type: *boolean*	Default: *true*

The SMTP protocol specification requires the client to wait for a response from the server at certain points in the dialogue. Without PIPELINING these synchronization points are after every command; with PIPELINING they are fewer, but they still exist.

Some spamming sites send out a complete set of SMTP commands without waiting for any response. Exim protects against this by rejecting a message if the client has sent further input when it should not have. The error response "554 SMTP synchronization error" is sent, and the connection is dropped. Testing for this error cannot be perfect because of transmission delays (unexpected input may be on its way but not yet received when Exim checks). However, it does detect many instances.

The check can be globally disabled by setting **smtp_enforce_sync** false. If you want to disable the check selectively (for example, only for certain hosts), you can do so by an appropriate use of a **control** modifier in an ACL (see section 40.20). See also **pipelining_advertise_hosts**.

smtp_etrn_command	Use: *main*	Type: *string†*	Default: *unset*

If this option is set, the given command is run whenever an SMTP ETRN command is received from a host that is permitted to issue such commands (see chapter 40). The string is split up into separate arguments which are independently expanded. The expansion variable *$domain* is set to the argument of the ETRN command, and no syntax checking is done on it. For example:

```
smtp_etrn_command = /etc/etrn_command $domain \
                    $sender_host_address
```

A new process is created to run the command, but Exim does not wait for it to complete. Consequently, its status cannot be checked. If the command cannot be run, a line is written to the panic log, but the ETRN caller still receives a 250 success response. Exim is normally running under its own uid when receiving SMTP, so it is not possible for it to change the uid before running the command.

smtp_etrn_serialize	Use: *main*	Type: *boolean*	Default: *true*

When this option is set, it prevents the simultaneous execution of more than one identical command as a result of ETRN in an SMTP connection. See section 45.8 for details.

smtp_load_reserve	Use: *main*	Type: *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**. If **smtp_reserve_hosts** is not set, no incoming SMTP calls are accepted when the load is over the limit. The option has no effect on ancient operating systems on which Exim cannot determine the load average. See also **deliver_queue_load_max** and **queue_only_load**.

smtp_max_synprot_errors	Use: *main*	Type: *integer*	Default: *3*

Exim rejects SMTP commands that contain syntax or protocol errors. In particular, a syntactically invalid email address, as in this command:

```
RCPT TO:<abc xyz@a.b.c>
```

causes immediate rejection of the command, before any other tests are done. (The ACL cannot be run if there is no valid address to set up for it.) An example of a protocol error is receiving RCPT before MAIL. If there are too many syntax or protocol errors in one SMTP session, the connection is dropped. The limit is set by this option.

When the PIPELINING extension to SMTP is in use, some protocol errors are "expected", for instance, a RCPT command after a rejected MAIL command. Exim assumes that PIPELINING will be used if it advertises it (see **pipelining_advertise_hosts**), and in this situation, "expected" errors do not count towards the limit.

smtp_max_unknown_commands	Use: *main*	Type: *integer*	Default: *3*

If there are too many unrecognized commands in an incoming SMTP session, an Exim server drops the connection. This is a defence against some kinds of abuse that subvert web clients into making connections to SMTP ports; in these circumstances, a number of non-SMTP command lines are sent first.

smtp_ratelimit_hosts	Use: *main*	Type: *host list*†	Default: *unset*

Some sites find it helpful to be able to limit the rate at which certain hosts can send them messages, and the rate at which an individual message can specify recipients.

Exim has two rate-limiting facilities. This section describes the older facility, which can limit rates within a single connection. The newer **ratelimit** ACL condition can limit rates across all connections. See section 40.35 for details of the newer facility.

When a host matches **smtp_ratelimit_hosts**, the values of **smtp_ratelimit_mail** and **smtp_ratelimit_rcpt** are used to control the rate of acceptance of MAIL and RCPT commands in a single SMTP session, respectively. Each option, if set, must contain a set of four comma-separated values:

- A threshold, before which there is no rate limiting.
- An initial time delay. Unlike other times in Exim, numbers with decimal fractional parts are allowed here.
- A factor by which to increase the delay each time.
- A maximum value for the delay. This should normally be less than 5 minutes, because after that time, the client is liable to timeout the SMTP command.

For example, these settings have been used successfully at the site which first suggested this feature, for controlling mail from their customers:

```
smtp_ratelimit_mail = 2,0.5s,1.05,4m
smtp_ratelimit_rcpt = 4,0.25s,1.015,4m
```

The first setting specifies delays that are applied to MAIL commands after two have been received over a single connection. The initial delay is 0.5 seconds, increasing by a factor of 1.05 each time. The second setting applies delays to RCPT commands when more than four occur in a single message.

smtp_ratelimit_mail	Use: *main*	Type: *string*	Default: *unset*

See **smtp_ratelimit_hosts** above.

smtp_ratelimit_rcpt	Use: *main*	Type: *string*	Default: *unset*

See **smtp_ratelimit_hosts** above.

smtp_receive_timeout	Use: *main*	Type: *time*	Default: *5m*

This sets a timeout value for SMTP reception. It applies to all forms of SMTP input, including batch SMTP. 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. A line is written to the log containing one of the following messages:

```
SMTP command timeout on connection from...
SMTP data timeout on connection from...
```

The former means that Exim was expecting to read an SMTP command; the latter means that it was in the DATA phase, reading the contents of a message.

The value set by this option can be overridden by the **-os** command-line option. A setting of zero time disables the timeout, but this should never be used for SMTP over TCP/IP. (It can be useful in some cases of local input using **-bs** or **-bS**.) For non-SMTP input, the reception timeout is controlled by **receive_timeout** and **-or**.

smtp_reserve_hosts	Use: *main*	Type: *host list*†	Default: *unset*

This option defines hosts for which SMTP connections are reserved; see **smtp_accept_reserve** and **smtp_load_reserve** above.

smtp_return_error_details	Use: *main*	Type: *boolean*	Default: *false*

In the default state, Exim uses bland messages such as "Administrative prohibition" when it rejects SMTP commands for policy reasons. Many sysadmins like this because it gives away little information to spammers. However, some other sysadmins who are applying strict checking policies want to give out much fuller information about failures. Setting **smtp_return_error_details** true causes Exim to be more forthcoming. For example, instead of "Administrative prohibition", it might give:

```
550-Rejected after DATA: '>' missing at end of address:
550 failing address in "From" header is: <user@dom.ain
```

spamd_address	Use: *main*	Type: *string*	Default: *see below*

This option is available when Exim is compiled with the content-scanning extension. It specifies how Exim connects to SpamAssassin's **spamd** daemon. The default value is

```
127.0.0.1 783
```

See section 41.2 for more details.

split_spool_directory	Use: *main*	Type: *boolean*	Default: *false*

If this option is set, it causes Exim to split its input directory into 62 subdirectories, each with a single alphanumeric character as its name. The sixth character of the message id is used to allocate messages to subdirectories; this is the least significant base-62 digit of the time of arrival of the message.

Splitting up the spool in this way may provide better performance on systems where there are long mail queues, by reducing the number of files in any one directory. The msglog directory is also split up in a similar way to the input directory; however, if **preserve_message_logs** is set, all old msglog files are still placed in the single directory *msglog.OLD*.

It is not necessary to take any special action for existing messages when changing **split_spool_directory**. Exim notices messages that are in the "wrong" place, and continues to process them. If the option is turned off after a period of being on, the subdirectories will eventually empty and be automatically deleted.

When **split_spool_directory** is set, the behaviour of queue runner processes changes. Instead of creating a list of all messages in the queue, and then trying to deliver each one in turn, it constructs a list of those in one sub-directory and tries to deliver them, before moving on to the next sub-directory. The sub-directories are processed in a random order. This spreads out the scanning of the input directories, and uses less memory. It is particularly beneficial when there are lots of messages on the queue. However, if **queue_run_in_order** is set, none of this new processing happens. The entire queue has to be scanned and sorted before any deliveries can start.

spool_directory	Use: *main*	Type: *string*†	Default: *set at compile time*

This defines the directory in which Exim keeps its spool, that is, the messages it is waiting to deliver. The default value is taken from the compile-time configuration setting, if there is one. If not, this option must be set. The string is expanded, so it can contain, for example, a reference to *$primary_hostname*.

If the spool directory name is fixed on your installation, it is recommended that you set it at build time rather than from this option, particularly if the log files are being written to the spool directory (see **log_file_path**). Otherwise log files cannot be used for errors that are detected early on, such as failures in the configuration file.

By using this option to override the compiled-in path, it is possible to run tests of Exim without using the standard spool.

sqlite_lock_timeout	Use: *main*	Type: *time*	Default: *5s*

This option controls the timeout that the *sqlite* lookup uses when trying to access an SQLite database. See section 9.25 for more details.

strict_acl_vars	Use: *main*	Type: *boolean*	Default: *false*

This option controls what happens if a syntactically valid but undefined ACL variable is referenced. If it is false (the default), an empty string is substituted; if it is true, an error is generated. See section 40.17 for details of ACL variables.

strip_excess_angle_brackets	Use: *main*	Type: *boolean*	Default: *false*

If this option is set, redundant pairs of angle brackets round "route-addr" items in addresses are stripped. For example, *<<xxx@a.b.c.d>>* is treated as *<xxx@a.b.c.d>*. If this is in the envelope and the message is passed on to another MTA, the excess angle brackets are not passed on. If this option is not set, multiple pairs of angle brackets cause a syntax error.

strip_trailing_dot	Use: *main*	Type: *boolean*	Default: *false*

If this option is set, a trailing dot at the end of a domain in an address is ignored. If this is in the envelope and the message is passed on to another MTA, the dot is not passed on. If this option is not set, a dot at the end of a domain causes a syntax error. However, addresses in header lines are checked only when an ACL requests header syntax checking.

syslog_duplication	Use: *main*	Type: *boolean*	Default: *true*

When Exim is logging to syslog, it writes the log lines for its three separate logs at different syslog priorities so that they can in principle be separated on the logging hosts. Some installations do not require this separation, and in those cases, the duplication of certain log lines is a nuisance. If **syslog_duplication** is set false, only one copy of any particular log line is written to syslog. For lines that normally go to both the main log and the reject log, the reject log version (possibly containing message header lines) is written, at LOG_NOTICE priority. Lines that normally go to both the main and the panic log are written at the LOG_ALERT priority.

syslog_facility	Use: *main*	Type: *string*	Default: *unset*

This option sets the syslog "facility" name, used when Exim is logging to syslog. The value must be one of the strings "mail", "user", "news", "uucp", "daemon", or "local*x*" where *x* is a digit between 0 and 7. If this option is unset, "mail" is used. See chapter 49 for details of Exim's logging.

syslog_processname	Use: *main*	Type: *string*	Default: `exim`

This option sets the syslog "ident" name, used when Exim is logging to syslog. The value must be no longer than 32 characters. See chapter 49 for details of Exim's logging.

syslog_timestamp	Use: *main*	Type: *boolean*	Default: *true*

If **syslog_timestamp** is set false, the timestamps on Exim's log lines are omitted when these lines are sent to syslog. See chapter 49 for details of Exim's logging.

system_filter	Use: *main*	Type: *string*†	Default: *unset*

This option specifies an Exim filter file that is applied to all messages at the start of each delivery attempt, before any routing is done. System filters must be Exim filters; they cannot be Sieve filters. If the system filter generates any deliveries to files or pipes, or any new mail messages, the appropriate **system_filter_..._transport** option(s) must be set, to define which transports are to be used. Details of this facility are given in chapter 43.

system_filter_directory_transport	Use: *main*	Type: *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. During the delivery, the variable *$address_file* contains the path name.

system_filter_file_transport	Use: *main*	Type: *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 "/". During the delivery, the variable *$address_file* contains the path name.

system_filter_group	Use: *main*	Type: *string*	Default: *unset*

This option is used only when **system_filter_user** is also set. It sets the gid under which the system filter is run, overriding any gid that is associated with the user. The value may be numerical or symbolic.

system_filter_pipe_transport	Use: *main*	Type: *string*†	Default: *unset*

This specifies the transport driver that is to be used when a **pipe** command is used in a system filter. During the delivery, the variable *$address_pipe* contains the pipe command.

system_filter_reply_transport	Use: *main*	Type: *string*†	Default: *unset*

This specifies the transport driver that is to be used when a **mail** command is used in a system filter.

system_filter_user	Use: *main*	Type: *string*	Default: *unset*

If this option is not set, the system filter is run in the main Exim delivery process, as root. When the option is set, the system filter runs in a separate process, as the given user. Unless the string consists entirely of digits, it is looked up in the password data. Failure to find the named user causes a configuration error. The gid is either taken from the password data, or specified by **system_filter_group**. When the uid is specified numerically, **system_filter_group** is required to be set.

If the system filter generates any pipe, file, or reply deliveries, the uid under which the filter is run is used when transporting them, unless a transport option overrides. Normally you should set **system_filter_user** if your system filter generates these kinds of delivery.

tcp_nodelay	Use: *main*	Type: *boolean*	Default: *true*

If this option is set false, it stops the Exim daemon setting the TCP_NODELAY option on its listening sockets. Setting TCP_NODELAY turns off the "Nagle algorithm", which is a way of improving network performance in interactive (character-by-character) situations. Turning it off should improve Exim's performance a bit, so that is what happens by default. However, it appears that some broken clients cannot cope, and time out. Hence this option. It affects only those sockets that are set up for listening by the daemon. Sockets created by the smtp transport for delivering mail always set TCP_NODELAY.

timeout_frozen_after	Use: *main*	Type: *time*	Default: *0s*

If **timeout_frozen_after** is set to a time greater than zero, a frozen message of any kind that has been on the queue for longer than the given time is automatically cancelled at the next queue run. If the frozen message is a bounce message, it is just discarded; otherwise, a bounce is sent to the sender, in a

similar manner to cancellation by the **-Mg** command line option. If you want to timeout frozen bounce messages earlier than other kinds of frozen message, see **ignore_bounce_errors_after**.

Note: the default value of zero means no timeouts; with this setting, frozen messages remain on the queue forever (except for any frozen bounce messages that are released by **ignore_bounce_errors_after**).

timezone	Use: *main*	Type: *string*	Default: *unset*

The value of **timezone** is used to set the environment variable TZ while running Exim (if it is different on entry). This ensures that all timestamps created by Exim are in the required timezone. If you want all your timestamps to be in UTC (aka GMT) you should set

```
timezone = UTC
```

The default value is taken from TIMEZONE_DEFAULT in *Local/Makefile*, or, if that is not set, from the value of the TZ environment variable when Exim is built. If **timezone** is set to the empty string, either at build or run time, any existing TZ variable is removed from the environment when Exim runs. This is appropriate behaviour for obtaining wall-clock time on some, but unfortunately not all, operating systems.

tls_advertise_hosts	Use: *main*	Type: *host list*†	Default: *unset*

When Exim is built with support for TLS encrypted connections, the availability of the STARTTLS command to set up an encrypted session is advertised in response to EHLO only to those client hosts that match this option. See chapter 39 for details of Exim's support for TLS.

tls_certificate	Use: *main*	Type: *string*†	Default: *unset*

The value of this option is expanded, and must then be the absolute path to a file which contains the server's certificates. The server's private key is also assumed to be in this file if **tls_privatekey** is unset. See chapter 39 for further details.

Note: The certificates defined by this option are used only when Exim is receiving incoming messages as a server. If you want to supply certificates for use when sending messages as a client, you must set the **tls_certificate** option in the relevant *smtp* transport.

tls_crl	Use: *main*	Type: *string*†	Default: *unset*

This option specifies a certificate revocation list. The expanded value must be the name of a file that contains a CRL in PEM format.

tls_dhparam	Use: *main*	Type: *string*†	Default: *unset*

The value of this option is expanded, and must then be the absolute path to a file which contains the server's DH parameter values. This is used only for OpenSSL. When Exim is linked with GnuTLS, this option is ignored. See section 39.2 for further details.

tls_on_connect_ports	Use: *main*	Type: *string list*	Default: *unset*

This option specifies a list of incoming SSMTP (aka SMTPS) ports that should operate the obsolete SSMTP (SMTPS) protocol, where a TLS session is immediately set up without waiting for the client to issue a STARTTLS command. For further details, see section 13.4.

tls_privatekey	Use: *main*	Type: *string*†	Default: *unset*

The value of this option is expanded, and must then be the absolute path to a file which contains the server's private key. If this option is unset, or if the expansion is forced to fail, or the result is an empty string, the private key is assumed to be in the same file as the server's certificates. See chapter 39 for further details.

tls_remember_esmtp	Use: *main*	Type: *boolean*	Default: *false*

If this option is set true, Exim violates the RFCs by remembering that it is in "esmtp" state after successfully negotiating a TLS session. This provides support for broken clients that fail to send a new EHLO after starting a TLS session.

tls_require_ciphers	Use: *main*	Type: *string*†	Default: *unset*

This option controls which ciphers can be used for incoming TLS connections. The *smtp* transport has an option of the same name for controlling outgoing connections. This option is expanded for each connection, so can be varied for different clients if required. The value of this option must be a list of permitted cipher suites. The OpenSSL and GnuTLS libraries handle cipher control in somewhat different ways. If GnuTLS is being used, the client controls the preference order of the available ciphers. Details are given in sections 39.4 and 39.5.

tls_try_verify_hosts	Use: *main*	Type: *host list*†	Default: *unset*

See **tls_verify_hosts** below.

tls_verify_certificates	Use: *main*	Type: *string*†	Default: *unset*

The value of this option is expanded, and must then be the absolute path to a file containing permitted certificates for clients that match **tls_verify_hosts** or **tls_try_verify_hosts**. Alternatively, if you are using OpenSSL, you can set **tls_verify_certificates** to the name of a directory containing certificate files. This does not work with GnuTLS; the option must be set to the name of a single file if you are using GnuTLS.

tls_verify_hosts	Use: *main*	Type: *host list*†	Default: *unset*

This option, along with **tls_try_verify_hosts**, controls the checking of certificates from clients. The expected certificates are defined by **tls_verify_certificates**, which must be set. A configuration error occurs if either **tls_verify_hosts** or **tls_try_verify_hosts** is set and **tls_verify_certificates** is not set.

Any client that matches **tls_verify_hosts** is constrained by **tls_verify_certificates**. When the client initiates a TLS session, it must present one of the listed certificates. If it does not, the connection is aborted. **Warning**: Including a host in **tls_verify_hosts** does not require the host to use TLS. It can still send SMTP commands through unencrypted connections. Forcing a client to use TLS has to be done separately using an ACL to reject inappropriate commands when the connection is not encrypted.

A weaker form of checking is provided by **tls_try_verify_hosts**. If a client matches this option (but not **tls_verify_hosts**), Exim requests a certificate and checks it against **tls_verify_certificates**, but does not abort the connection if there is no certificate or if it does not match. This state can be detected in an ACL, which makes it possible to implement policies such as "accept for relay only if a verified certificate has been received, but accept for local delivery if encrypted, even without a verified certificate".

Client hosts that match neither of these lists are not asked to present certificates.

trusted_groups	Use: *main*	Type: *string list*†	Default: *unset*

This option is expanded just once, at the start of Exim's processing. If this option is set, any process that is running in one of the listed groups, or which has one of them as a supplementary group, is trusted. The groups can be specified numerically or by name. See section 5.2 for details of what trusted callers are permitted to do. If neither **trusted_groups** nor **trusted_users** is set, only root and the Exim user are trusted.

trusted_users	Use: *main*	Type: *string list*†	Default: *unset*

This option is expanded just once, at the start of Exim's processing. If this option is set, any process that is running as one of the listed users is trusted. The users can be specified numerically or by name. See section 5.2 for details of what trusted callers are permitted to do. If neither **trusted_groups** nor **trusted_users** is set, only root and the Exim user are trusted.

unknown_login	Use: *main*	Type: *string*†	Default: *unset*

This is a specialized feature for use in unusual configurations. By default, if the uid of the caller of Exim cannot be looked up using *getpwuid()*, Exim gives up. The **unknown_login** option can be used to set a login name to be used in this circumstance. It is expanded, so values like **user$caller_uid** can be set. When **unknown_login** is used, the value of **unknown_username** is used for the user's real name (gecos field), unless this has been set by the **-F** option.

unknown_username	Use: *main*	Type: *string*	Default: *unset*

See **unknown_login**.

untrusted_set_sender	Use: *main*	Type: *address list*†	Default: *unset*

When an untrusted user submits a message to Exim using the standard input, Exim normally creates an envelope sender address from the user's login and the default qualification domain. Data from the **-f** option (for setting envelope senders on non-SMTP messages) or the SMTP MAIL command (if **-bs** or **-bS** is used) is ignored.

However, untrusted users are permitted to set an empty envelope sender address, to declare that a message should never generate any bounces. For example:

```
exim -f '<>' user@domain.example
```

The **untrusted_set_sender** option allows you to permit untrusted users to set other envelope sender addresses in a controlled way. When it is set, untrusted users are allowed to set envelope sender addresses that match any of the patterns in the list. Like all address lists, the string is expanded. The identity of the user is in *$sender_ident*, so you can, for example, restrict users to setting senders that start with their login ids followed by a hyphen by a setting like this:

```
untrusted_set_sender = ^$sender_ident-
```

If you want to allow untrusted users to set envelope sender addresses without restriction, you can use

```
untrusted_set_sender = *
```

The **untrusted_set_sender** option applies to all forms of local input, but only to the setting of the envelope sender. It does not permit untrusted users to use the other options which trusted user can use to override message parameters. Furthermore, it does not stop Exim from removing an existing *Sender:* header in the message, or from adding a *Sender:* header if necessary. See **local_sender_retain** and **local_from_check** for ways of overriding these actions. The handling of the *Sender:* header is also described in section 44.16.

The log line for a message's arrival shows the envelope sender following "<=". For local messages, the user's login always follows, after "U=". In **-bp** displays, and in the Exim monitor, if an untrusted user sets an envelope sender address, the user's login is shown in parentheses after the sender address.

uucp_from_pattern	Use: *main*	Type: *string*	Default: *see below*

Some applications that pass messages to an MTA via a command line interface use an initial line starting with "From " to pass the envelope sender. In particular, this is used by UUCP software. Exim recognizes such a line by means of a regular expression that is set in **uucp_from_pattern**. When the pattern matches, the sender address is constructed by expanding the contents of **uucp_from_sender**, provided that the caller of Exim is a trusted user. The default pattern recognizes lines in the following two forms:

```
From ph10 Fri Jan  5 12:35 GMT 1996
From ph10 Fri, 7 Jan 97 14:00:00 GMT
```

The pattern can be seen by running

```
exim -bP uucp_from_pattern
```

It checks only up to the hours and minutes, and allows for a 2-digit or 4-digit year in the second case. The first word after "From " is matched in the regular expression by a parenthesized subpattern. The default value for **uucp_from_sender** is "$1", which therefore just uses this first word ("ph10" in the example above) as the message's sender. See also **ignore_fromline_hosts**.

uucp_from_sender	Use: *main*	Type: *string*†	Default: *$1*

See **uucp_from_pattern** above.

warn_message_file	Use: *main*	Type: *string*	Default: *unset*

This option defines a template file containing paragraphs of text to be used for constructing the warning message which is sent by Exim when a message has been on the queue for a specified amount of time, as specified by **delay_warning**. Details of the file's contents are given in chapter 46. See also **bounce_message_file**.

write_rejectlog	Use: *main*	Type: *boolean*	Default: *true*

If this option is set false, Exim no longer writes anything to the reject log. See chapter 49 for details of what Exim writes to its logs.

15. Generic options for routers

This chapter describes the generic options that apply to all routers. Those that are preconditions are marked with ‡ in the "use" field.

For a general description of how a router operates, see sections 3.10 and 3.12. The latter specifies the order in which the preconditions are tested. The order of expansion of the options that provide data for a transport is: **errors_to**, **headers_add**, **headers_remove**, **transport**.

address_data	Use: *routers*	Type: *string*†	Default: *unset*

The string is expanded just before the router is run, that is, after all the precondition tests have succeeded. If the expansion is forced to fail, the router declines, the value of **address_data** remains unchanged, and the **more** option controls what happens next. Other expansion failures cause delivery of the address to be deferred.

When the expansion succeeds, the value is retained with the address, and can be accessed using the variable *$address_data* in the current router, subsequent routers, and the eventual transport.

Warning: If the current or any subsequent router is a *redirect* router that runs a user's filter file, the contents of *$address_data* are accessible in the filter. This is not normally a problem, because such data is usually either not confidential or it "belongs" to the current user, but if you do put confidential data into *$address_data* you need to remember this point.

Even if the router declines or passes, the value of *$address_data* remains with the address, though it can be changed by another **address_data** setting on a subsequent router. If a router generates child addresses, the value of *$address_data* propagates to them. This also applies to the special kind of "child" that is generated by a router with the **unseen** option.

The idea of **address_data** is that you can use it to look up a lot of data for the address once, and then pick out parts of the data later. For example, you could use a single LDAP lookup to return a string of the form

```
uid=1234 gid=5678 mailbox=/mail/xyz forward=/home/xyz/.forward
```

In the transport you could pick out the mailbox by a setting such as

```
file = ${extract{mailbox}{$address_data}}
```

This makes the configuration file less messy, and also reduces the number of lookups (though Exim does cache lookups).

The **address_data** facility is also useful as a means of passing information from one router to another, and from a router to a transport. In addition, if *$address_data* is set by a router when verifying a recipient address from an ACL, it remains available for use in the rest of the ACL statement. After verifying a sender, the value is transferred to *$sender_address_data*.

address_test	Use: *routers*‡	Type: *boolean*	Default: *true*

If this option is set false, the router is skipped when routing is being tested by means of the **-bt** command line option. This can be a convenience when your first router sends messages to an external scanner, because it saves you having to set the "already scanned" indicator when testing real address routing.

cannot_route_message	Use: *routers*	Type: *string*†	Default: *unset*

This option specifies a text message that is used when an address cannot be routed because Exim has run out of routers. The default message is "Unrouteable address". This option is useful only on routers that have **more** set false, or on the very last router in a configuration, because the value that is used is taken from the last router that is considered. This includes a router that is skipped because its

preconditions are not met, as well as a router that declines. For example, using the default configuration, you could put:

```
cannot_route_message = Remote domain not found in DNS
```

on the first router, which is a *dnslookup* router with **more** set false, and

```
cannot_route_message = Unknown local user
```

on the final router that checks for local users. If string expansion fails for this option, the default message is used. Unless the expansion failure was explicitly forced, a message about the failure is written to the main and panic logs, in addition to the normal message about the routing failure.

caseful_local_part	Use: *routers*	Type: *boolean*	Default: *false*

By default, routers handle the local parts of addresses in a case-insensitive manner, though the actual case is preserved for transmission with the message. If you want the case of letters to be significant in a router, you must set this option true. For individual router options that contain address or local part lists (for example, **local_parts**), case-sensitive matching can be turned on by "+caseful" as a list item. See section 10.20 for more details.

The value of the *$local_part* variable is forced to lower case while a router is running unless **caseful_local_part** is set. When a router assigns an address to a transport, the value of *$local_part* when the transport runs is the same as it was in the router. Similarly, when a router generates child addresses by aliasing or forwarding, the values of *$original_local_part* and *$parent_local_part* are those that were used by the redirecting router.

This option applies to the processing of an address by a router. When a recipient address is being processed in an ACL, there is a separate **control** modifier that can be used to specify case-sensitive processing within the ACL (see section 40.20).

check_local_user	Use: *routers*‡	Type: *boolean*	Default: *false*

When this option is true, Exim checks that the local part of the recipient address (with affixes removed if relevant) is the name of an account on the local system. The check is done by calling the *getpwnam()* function rather than trying to read */etc/passwd* directly. This means that other methods of holding password data (such as NIS) are supported. If the local part is a local user, *$home* is set from the password data, and can be tested in other preconditions that are evaluated after this one (the order of evaluation is given in section 3.12). However, the value of *$home* can be overridden by **router_home_directory**. If the local part is not a local user, the router is skipped.

If you want to check that the local part is either the name of a local user or matches something else, you cannot combine **check_local_user** with a setting of **local_parts**, because that specifies the logical *and* of the two conditions. However, you can use a *passwd* lookup in a **local_parts** setting to achieve this. For example:

```
local_parts = passwd;$local_part : lsearch;/etc/other/users
```

Note, however, that the side effects of **check_local_user** (such as setting up a home directory) do not occur when a *passwd* lookup is used in a **local_parts** (or any other) precondition.

condition	Use: *routers*‡	Type: *string*†	Default: *unset*

This option specifies a general precondition test that has to succeed for the router to be called. The **condition** option is the last precondition to be evaluated (see section 3.12). The string is expanded, and if the result is a forced failure, or an empty string, or one of the strings "0" or "no" or "false" (checked without regard to the case of the letters), the router is skipped, and the address is offered to the next one.

If the result is any other value, the router is run (as this is the last precondition to be evaluated, all the other preconditions must be true).

The **condition** option provides a means of applying custom conditions to the running of routers. Note that in the case of a simple conditional expansion, the default expansion values are exactly what is wanted. For example:

```
condition = ${if >{$message_age}{600}}
```

Because of the default behaviour of the string expansion, this is equivalent to

```
condition = ${if >{$message_age}{600}{true}{}}
```

If the expansion fails (other than forced failure) delivery is deferred. Some of the other precondition options are common special cases that could in fact be specified using **condition**.

debug_print	Use: *routers*	Type: *string*†	Default: *unset*

If this option is set and debugging is enabled (see the **-d** command line option), the string is expanded and included in the debugging output. If expansion of the string fails, the error message is written to the debugging output, and Exim carries on processing. This option is provided to help with checking out the values of variables and so on when debugging router configurations. For example, if a **condition** option appears not to be working, **debug_print** can be used to output the variables it references. The output happens after checks for **domains**, **local_parts**, and **check_local_user** but before any other preconditions are tested. A newline is added to the text if it does not end with one.

disable_logging	Use: *routers*	Type: *boolean*	Default: *false*

If this option is set true, nothing is logged for any routing errors or for any deliveries caused by this router. You should not set this option unless you really, really know what you are doing. See also the generic transport option of the same name.

domains	Use: *routers*‡	Type: *domain list*†	Default: *unset*

If this option is set, the router is skipped unless the current domain matches 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 *$domain_data* for use in string expansions of the driver's private options. See section 3.12 for a list of the order in which preconditions are evaluated.

driver	Use: *routers*	Type: *string*	Default: *unset*

This option must always be set. It specifies which of the available routers is to be used.

errors_to	Use: *routers*	Type: *string*†	Default: *unset*

If a router successfully handles an address, it may assign the address to a transport for delivery or it may generate child addresses. In both cases, if there is a delivery problem during later processing, the resulting bounce message is sent to the address that results from expanding this string, provided that the address verifies successfully. The **errors_to** option is expanded before **headers_add**, **headers_remove**, and **transport**.

The **errors_to** setting associated with an address can be overridden if it subsequently passes through other routers that have their own **errors_to** settings, or if the message is delivered by a transport with a **return_path** setting.

If **errors_to** is unset, or the expansion is forced to fail, or the result of the expansion fails to verify, the errors address associated with the incoming address is used. At top level, this is the envelope sender. A non-forced expansion failure causes delivery to be deferred.

If an address for which **errors_to** has been set ends up being delivered over SMTP, the envelope sender for that delivery is the **errors_to** value, so that any bounces that are generated by other MTAs

on the delivery route are also sent there. You can set **errors_to** to the empty string by either of these settings:

```
errors_to =
errors_to = ""
```

An expansion item that yields an empty string has the same effect. If you do this, a locally detected delivery error for addresses processed by this router no longer gives rise to a bounce message; the error is discarded. If the address is delivered to a remote host, the return path is set to <>, unless overridden by the **return_path** option on the transport.

If for some reason you want to discard local errors, but use a non-empty MAIL command for remote delivery, you can preserve the original return path in *$address_data* in the router, and reinstate it in the transport by setting **return_path**.

The most common use of **errors_to** is to direct mailing list bounces to the manager of the list, as described in section 47.2, or to implement VERP (Variable Envelope Return Paths) (see section 47.6).

expn	Use: *routers*‡	Type: *boolean*	Default: *true*

If this option is turned off, the router is skipped when testing an address as a result of processing an SMTP EXPN command. You might, for example, want to turn it off on a router for users' *.forward* files, while leaving it on for the system alias file. See section 3.12 for a list of the order in which preconditions are evaluated.

The use of the SMTP EXPN command is controlled by an ACL (see chapter 40). When Exim is running an EXPN command, it is similar to testing an address with **-bt**. Compare VRFY, whose counterpart is **-bv**.

fail_verify	Use: *routers*	Type: *boolean*	Default: *false*

Setting this option has the effect of setting both **fail_verify_sender** and **fail_verify_recipient** to the same value.

fail_verify_recipient	Use: *routers*	Type: *boolean*	Default: *false*

If this option is true and an address is accepted by this router when verifying a recipient, verification fails.

fail_verify_sender	Use: *routers*	Type: *boolean*	Default: *false*

If this option is true and an address is accepted by this router when verifying a sender, verification fails.

fallback_hosts	Use: *routers*	Type: *string list*	Default: *unset*

String expansion is not applied to this option. The argument must be a colon-separated list of host names or IP addresses. The list separator can be changed (see section 6.19), and a port can be specified with each name or address. In fact, the format of each item is exactly the same as defined for the list of hosts in a *manualroute* router (see section 20.5).

If a router 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. If **hosts_randomize** is set on the transport, the order of the list is randomized for each use. See the **fallback_hosts** option of the *smtp* transport for further details.

group	Use: *routers*	Type: *string*†	Default: *see below*

When a router queues an address for a transport, and the transport does not specify a group, the group given here is used when running the delivery process. The group may be specified numerically or by name. If expansion fails, the error is logged and delivery is deferred. The default is unset, unless **check_local_user** is set, when the default is taken from the password information. See also **initgroups** and **user** and the discussion in chapter 23.

headers_add	Use: *routers*	Type: *string*†	Default: *unset*

This option specifies a string of text that is expanded at routing time, and associated with any addresses that are accepted by the router. However, this option has no effect when an address is just being verified. The way in which the text is used to add header lines at transport time is described in section 44.17. New header lines are not actually added until the message is in the process of being transported. This means that references to header lines in string expansions in the transport's configuration do not "see" the added header lines.

The **headers_add** option is expanded after **errors_to**, but before **headers_remove** and **transport**. 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.

Warning 1: The **headers_add** option cannot be used for a *redirect* router that has the **one_time** option set.

Warning 2: If the **unseen** option is set on the router, all header additions are deleted when the address is passed on to subsequent routers. For a **redirect** router, if a generated address is the same as the incoming address, this can lead to duplicate addresses with different header modifications. Exim does not do duplicate deliveries (except, in certain circumstances, to pipes -- see section 22.7), but it is undefined which of the duplicates is discarded, so this ambiguous situation should be avoided. The **repeat_use** option of the **redirect** router may be of help.

headers_remove	Use: *routers*	Type: *string*†	Default: *unset*

This option specifies a string of text that is expanded at routing time, and associated with any addresses that are accepted by the router. However, this option has no effect when an address is just being verified. The way in which the text is used to remove header lines at transport time is described in section 44.17. Header lines are not actually removed until the message is in the process of being transported. This means that references to header lines in string expansions in the transport's configuration still "see" the original header lines.

The **headers_remove** option is expanded after **errors_to** and **headers_add**, but before **transport**. If the expansion is forced to fail, the option has no effect. Other expansion failures are treated as configuration errors.

Warning 1: The **headers_remove** option cannot be used for a *redirect* router that has the **one_time** option set.

Warning 2: If the **unseen** option is set on the router, all header removal requests are deleted when the address is passed on to subsequent routers, and this can lead to problems with duplicates -- see the similar warning for **headers_add** above.

ignore_target_hosts	Use: *routers*	Type: *host list*†	Default: *unset*

Although this option is a host list, it should normally contain IP address entries rather than names. If any host that is looked up by the router has an IP address that matches an item in this list, Exim behaves as if that IP address did not exist. This option allows you to cope with rogue DNS entries like

```
remote.domain.example.  A  127.0.0.1
```

by setting

```
ignore_target_hosts = 127.0.0.1
```

on the relevant router. If all the hosts found by a *dnslookup* router are discarded in this way, the router declines. In a conventional configuration, an attempt to mail to such a domain would normally provoke the "unrouteable domain" error, and an attempt to verify an address in the domain would fail. Similarly, if **ignore_target_hosts** is set on an *ipliteral* router, the router declines if presented with one of the listed addresses.

You can use this option to disable the use of IPv4 or IPv6 for mail delivery by means of the first or the second of the following settings, respectively:

```
ignore_target_hosts = 0.0.0.0/0
ignore_target_hosts = <; 0::0/0
```

The pattern in the first line matches all IPv4 addresses, whereas the pattern in the second line matches all IPv6 addresses.

This option may also be useful for ignoring link-local and site-local IPv6 addresses. Because, like all host lists, the value of **ignore_target_hosts** is expanded before use as a list, it is possible to make it dependent on the domain that is being routed.

During its expansion, *$host_address* is set to the IP address that is being checked.

initgroups	Use: *routers*	Type: *boolean*	Default: *false*

If the router queues an address for a transport, and this option is true, and the uid supplied by the router 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 set up. See also **group** and **user** and the discussion in chapter 23.

local_part_prefix	Use: *routers*‡	Type: *string list*	Default: *unset*

If this option is set, the router is skipped unless the local part starts with one of the given strings, or **local_part_prefix_optional** is true. See section 3.12 for a list of the order in which preconditions are evaluated.

The list is scanned from left to right, and the first prefix that matches is used. A limited form of wildcard is available; if the prefix begins with an asterisk, it matches the longest possible sequence of arbitrary characters at the start of the local part. An asterisk should therefore always be followed by some character that does not occur in normal local parts. Wildcarding can be used to set up multiple user mailboxes, as described in section 47.8.

During the testing of the **local_parts** option, and while the router is running, the prefix is removed from the local part, and is available in the expansion variable *$local_part_prefix*. When a message is being delivered, if the router accepts the address, this remains true during subsequent delivery by a transport. In particular, the local part that is transmitted in the RCPT command for LMTP, SMTP, and BSMTP deliveries has the prefix removed by default. This behaviour can be overridden by setting **rcpt_include_affixes** true on the relevant transport.

When an address is being verified, **local_part_prefix** affects only the behaviour of the router. If the callout feature of verification is in use, this means that the full address, including the prefix, will be used during the callout.

The prefix facility is commonly used to handle local parts of the form **owner-something**. Another common use is to support local parts of the form **real-username** to bypass a user's *.forward* file – helpful when trying to tell a user their forwarding is broken – by placing a router like this one immediately before the router that handles *.forward* files:

```
real_localuser:
  driver = accept
  local_part_prefix = real-
```

```
check_local_user
transport = local_delivery
```

For security, it would probably be a good idea to restrict the use of this router to locally-generated messages, using a condition such as this:

```
condition = ${if match {$sender_host_address}\
                       {\N^(|127\.0\.0\.1)$\N}}
```

If both **local_part_prefix** and **local_part_suffix** are set for a router, both conditions must be met if not optional. Care must be taken if wildcards are used in both a prefix and a suffix on the same router. Different separator characters must be used to avoid ambiguity.

local_part_prefix_optional	Use: *routers*	Type: *boolean*	Default: *false*

See **local_part_prefix** above.

local_part_suffix	Use: *routers*‡	Type: *string list*	Default: *unset*

This option operates in the same way as **local_part_prefix**, except that the local part must end (rather than start) with the given string, the **local_part_suffix_optional** option determines whether the suffix is mandatory, and the wildcard * character, if present, must be the last character of the suffix. This option facility is commonly used to handle local parts of the form **something-request** and multiple user mailboxes of the form **username-foo**.

local_part_suffix_optional	Use: *routers*	Type: *boolean*	Default: *false*

See **local_part_suffix** above.

local_parts	Use: *routers*‡	Type: *local part list*†	Default: *unset*

The router is run only if the local part of the address matches the list. See section 3.12 for a list of the order in which preconditions are evaluated, and section 10.21 for a discussion of local part lists. Because the string is expanded, it is possible to make it depend on the domain, for example:

```
local_parts = dbm;/usr/local/specials/$domain
```

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 router's private options. You might use this option, for example, if you have a large number of local virtual domains, and you want to send all postmaster mail to the same place without having to set up an alias in each virtual domain:

```
postmaster:
  driver = redirect
  local_parts = postmaster
  data = postmaster@real.domain.example
```

log_as_local	Use: *routers*	Type: *boolean*	Default: *see below*

Exim has two logging styles for delivery, the idea being to make local deliveries stand out more visibly from remote ones. In the "local" style, the recipient address is given just as the local part, without a domain. The use of this style is controlled by this option. It defaults to true for the *accept* router, and false for all the others. This option applies only when a router assigns an address to a transport. It has no effect on routers that redirect addresses.

more	Use: *routers*	Type: *boolean*†	Default: *true*

The result of string expansion for this option must be a valid boolean value, that is, one of the strings "yes", "no", "true", or "false". Any other result causes an error, and delivery is deferred. If the expansion is forced to fail, the default value for the option (true) is used. Other failures cause delivery to be deferred.

If this option is set false, and the router declines to handle the address, no further routers are tried, routing fails, and the address is bounced. However, if the router explicitly passes an address to the following router by means of the setting

```
self = pass
```

or otherwise, the setting of **more** is ignored. Also, the setting of **more** does not affect the behaviour if one of the precondition tests fails. In that case, the address is always passed to the next router.

Note that **address_data** is not considered to be a precondition. If its expansion is forced to fail, the router declines, and the value of **more** controls what happens next.

pass_on_timeout	Use: *routers*	Type: *boolean*	Default: *false*

If a router times out during a host lookup, it normally causes deferral of the address. If **pass_on_timeout** is set, the address is 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 immediately be delivered.

There are occasional other temporary errors that can occur while doing DNS lookups. They are treated in the same way as a timeout, and this option applies to all of them.

pass_router	Use: *routers*	Type: *string*	Default: *unset*

Routers that recognize the generic **self** option (*dnslookup*, *ipliteral*, and *manualroute*) are able to return "pass", forcing routing to continue, and overriding a false setting of **more**. When one of these routers returns "pass", the address is normally handed on to the next router in sequence. This can be changed by setting **pass_router** to the name of another router. However (unlike **redirect_router**) the named router must be below the current router, to avoid loops. Note that this option applies only to the special case of "pass". It does not apply when a router returns "decline" because it cannot handle an address.

redirect_router	Use: *routers*	Type: *string*	Default: *unset*

Sometimes an administrator knows that it is pointless to reprocess addresses generated from alias or forward files with the same router 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 a large file.

The **redirect_router** option can be set to the name of any router instance. It causes the routing of any generated addresses to start at the named router instead of at the first router. This option has no effect if the router in which it is set does not generate new addresses.

require_files	Use: *routers*‡	Type: *string list*†	Default: *unset*

This option provides a general mechanism for predicating the running of a router on the existence or non-existence of certain files or directories. Before running a router, as one of its precondition tests, Exim works its way through the **require_files** list, expanding each item separately.

Because the list is split before expansion, any colons in expansion items must be doubled, or the facility for using a different list separator must be used. If any expansion is forced to fail, the item is ignored. Other expansion failures cause routing of the address to be deferred.

If any expanded string is empty, it is ignored. Otherwise, except as described below, each string must be a fully qualified file path, optionally preceded by "!". The paths are passed to the *stat()* function to test for the existence of the files or directories. The router is skipped if any paths not preceded by "!" do not exist, or if any paths preceded by "!" do exist.

If *stat()* cannot determine whether a file exists or not, delivery of the message is deferred. This can happen when NFS-mounted filesystems are unavailable.

This option is checked after the **domains**, **local_parts**, and **senders** options, so you cannot use it to check for the existence of a file in which to look up a domain, local part, or sender. (See section 3.12 for a full list of the order in which preconditions are evaluated.) However, as these options are all expanded, you can use the **exists** expansion condition to make such tests. The **require_files** option is intended for checking files that the router may be going to use internally, or which are needed by a transport (for example *.procmailrc*).

During delivery, the *stat()* function is run as root, but there is a facility for some checking of the accessibility of a file by another user. This is not a proper permissions check, but just a "rough" check that operates as follows:

If an item in a **require_files** list does not contain any forward slash characters, it is taken to be the user (and optional group, separated by a comma) to be checked for subsequent files in the list. If no group is specified but the user is specified symbolically, the gid associated with the uid is used. For example:

```
require_files = mail:/some/file
require_files = $local_part:$home/.procmailrc
```

If a user or group name in a **require_files** list does not exist, the **require_files** condition fails.

Exim performs the check by scanning along the components of the file path, and checking the access for the given uid and gid. It checks for "x" access on directories, and "r" access on the final file. Note that this means that file access control lists, if the operating system has them, are ignored.

Warning 1: When the router is being run to verify addresses for an incoming SMTP message, Exim is not running as root, but under its own uid. This may affect the result of a **require_files** check. In particular, *stat()* may yield the error EACCES ("Permission denied"). This means that the Exim user is not permitted to read one of the directories on the file's path.

Warning 2: Even when Exim is running as root while delivering a message, *stat()* can yield EACCES for a file in an NFS directory that is mounted without root access. In this case, if a check for access by a particular user is requested, Exim creates a subprocess that runs as that user, and tries the check again in that process.

The default action for handling an unresolved EACCES is to consider it to be caused by a configuration error, and routing is deferred because the existence or non-existence of the file cannot be determined. However, in some circumstances it may be desirable to treat this condition as if the file did not exist. If the file name (or the exclamation mark that precedes the file name for non-existence) is preceded by a plus sign, the EACCES error is treated as if the file did not exist. For example:

```
require_files = +/some/file
```

If the router is not an essential part of verification (for example, it handles users' *.forward* files), another solution is to set the **verify** option false so that the router is skipped when verifying.

retry_use_local_part	Use: *routers*	Type: *boolean*	Default: *see below*

When a delivery suffers a temporary routing failure, a retry record is created in Exim's hints database. For addresses whose routing depends only on the domain, the key for the retry record should not involve the local part, but for other addresses, both the domain and the local part should be included. Usually, remote routing is of the former kind, and local routing is of the latter kind.

This option controls whether the local part is used to form the key for retry hints for addresses that suffer temporary errors while being handled by this router. The default value is true for any router that

has **check_local_user** set, and false otherwise. Note that this option does not apply to hints keys for transport delays; they are controlled by a generic transport option of the same name.

The setting of **retry_use_local_part** applies only to the router on which it appears. If the router generates child addresses, they are routed independently; this setting does not become attached to them.

router_home_directory	Use: *routers*	Type: *string*†	Default: *unset*

This option sets a home directory for use while the router is running. (Compare **transport_home_directory**, which sets a home directory for later transporting.) In particular, if used on a *redirect* router, this option sets a value for *$home* while a filter is running. The value is expanded; forced expansion failure causes the option to be ignored – other failures cause the router to defer.

Expansion of **router_home_directory** happens immediately after the **check_local_user** test (if configured), before any further expansions take place. (See section 3.12 for a list of the order in which preconditions are evaluated.) While the router is running, **router_home_directory** overrides the value of *$home* that came from **check_local_user**.

When a router accepts an address and assigns it to a local transport (including the cases when a *redirect* router generates a pipe, file, or autoreply delivery), the home directory setting for the transport is taken from the first of these values that is set:

- The **home_directory** option on the transport;
- The **transport_home_directory** option on the router;
- The password data if **check_local_user** is set on the router;
- The **router_home_directory** option on the router.

In other words, **router_home_directory** overrides the password data for the router, but not for the transport.

self	Use: *routers*	Type: *string*	Default: *freeze*

This option applies to those routers that use a recipient address to find a list of remote hosts. Currently, these are the *dnslookup*, *ipliteral*, and *manualroute* routers. Certain configurations of the *queryprogram* router can also specify a list of remote hosts. Usually such routers are configured to send the message to a remote host via an *smtp* transport. The **self** option specifies what happens when the first host on the list turns out to be the local host. The way in which Exim checks for the local host is described in section 13.8.

Normally this situation indicates either an error in Exim's configuration (for example, the router should be configured not to process this domain), or an error in the DNS (for example, the MX should not point to this host). For this reason, the default action is to log the incident, defer the address, and freeze the message. The following alternatives are provided for use in special cases:

defer
Delivery of the message is tried again later, but the message is not frozen.

reroute: <*domain*>
The domain is changed to the given domain, and the address is passed back to be reprocessed by the routers. No rewriting of headers takes place. This behaviour is essentially a redirection.

reroute: rewrite: <*domain*>
The domain is changed to the given domain, and the address is passed back to be reprocessed by the routers. Any headers that contain the original domain are rewritten.

pass
The router passes the address to the next router, or to the router named in the **pass_router** option if it is set. This overrides **no_more**. During subsequent routing and delivery, the variable *$self_*

hostname contains the name of the local host that the router encountered. This can be used to distinguish between different cases for hosts with multiple names. The combination

```
self = pass
no_more
```

ensures that only those addresses that routed to the local host are passed on. Without **no_more**, addresses that were declined for other reasons would also be passed to the next router.

fail
Delivery fails and an error report is generated.

send
The anomaly is ignored and the address is queued for the transport. This setting should be used with extreme caution. For an *smtp* transport, it makes sense only in cases where the program that is listening on the SMTP port 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.

senders	Use: *routers*‡	Type: *address list*†	Default: *unset*

If this option is set, the router is skipped unless the message's sender address matches something on the list. See section 3.12 for a list of the order in which preconditions are evaluated.

There are issues concerning verification when the running of routers is dependent on the sender. When Exim is verifying the address in an **errors_to** setting, it sets the sender to the null string. When using the **-bt** option to check a configuration file, it is necessary also to use the **-f** option to set an appropriate sender. For incoming mail, the sender is unset when verifying the sender, but is available when verifying any recipients. If the SMTP VRFY command is enabled, it must be used after MAIL if the sender address matters.

translate_ip_address	Use: *routers*	Type: *string*†	Default: *unset*

There exist some rare networking situations (for example, packet radio) where it is helpful to be able to translate IP addresses generated by normal routing mechanisms into other IP addresses, thus performing a kind of manual IP routing. This should be done only if the normal IP routing of the TCP/IP stack is inadequate or broken. Because this is an extremely uncommon requirement, the code to support this option is not included in the Exim binary unless SUPPORT_TRANSLATE_IP_ADDRESS=yes is set in *Local/Makefile*.

The **translate_ip_address** string is expanded for every IP address generated by the router, with the generated address set in *$host_address*. If the expansion is forced to fail, no action is taken. For any other expansion error, delivery of the message is deferred. If the result of the expansion is an IP address, that replaces the original address; otherwise the result is assumed to be a host name – this is looked up using *gethostbyname()* (or *getipnodebyname()* when available) to produce one or more replacement IP addresses. For example, to subvert all IP addresses in some specific networks, this could be added to a router:

```
translate_ip_address = \
  ${lookup{${mask:$host_address/26}}lsearch{/some/file}\
    {$value}fail}}
```

The file would contain lines like

```
10.2.3.128/26    some.host
10.8.4.34/26     10.44.8.15
```

You should not make use of this facility unless you really understand what you are doing.

transport	Use: *routers*	Type: *string*†	Default: *unset*

This option specifies the transport to be used when a router accepts an address and sets it up for delivery. A transport is never needed if a router is used only for verification. The value of the option is expanded at routing time, after the expansion of **errors_to**, **headers_add**, and **headers_remove**, and result must be the name of one of the configured transports. If it is not, delivery is deferred.

The **transport** option is not used by the *redirect* router, but it does have some private options that set up transports for pipe and file deliveries (see chapter 22).

transport_current_directory	Use: *routers*	Type: *string*†	Default: *unset*

This option associates a current directory with any address that is routed to a local transport. This can happen either because a transport is explicitly configured for the router, or because it generates a delivery to a file or a pipe. During the delivery process (that is, at transport time), this option string is expanded and is set as the current directory, unless overridden by a setting on the transport. If the expansion fails for any reason, including forced failure, an error is logged, and delivery is deferred. See chapter 23 for details of the local delivery environment.

transport_home_directory	Use: *routers*	Type: *string*†	Default: *see below*

This option associates a home directory with any address that is routed to a local transport. This can happen either because a transport is explicitly configured for the router, or because it generates a delivery to a file or a pipe. During the delivery process (that is, at transport time), the option string is expanded and is set as the home directory, unless overridden by a setting of **home_directory** on the transport. If the expansion fails for any reason, including forced failure, an error is logged, and delivery is deferred.

If the transport does not specify a home directory, and **transport_home_directory** is not set for the router, the home directory for the transport is taken from the password data if **check_local_user** is set for the router. Otherwise it is taken from **router_home_directory** if that option is set; if not, no home directory is set for the transport.

See chapter 23 for further details of the local delivery environment.

unseen	Use: *routers*	Type: *boolean*†	Default: *false*

The result of string expansion for this option must be a valid boolean value, that is, one of the strings "yes", "no", "true", or "false". Any other result causes an error, and delivery is deferred. If the expansion is forced to fail, the default value for the option (false) is used. Other failures cause delivery to be deferred.

When this option is set true, routing does not cease if the router accepts the address. Instead, a copy of the incoming address is passed to the next router, overriding a false setting of **more**. There is little point in setting **more** false if **unseen** is always true, but it may be useful in cases when the value of **unseen** contains expansion items (and therefore, presumably, is sometimes true and sometimes false).

Setting the **unseen** option has a similar effect to the **unseen** command qualifier in filter files. It can be used to cause copies of messages to be delivered to some other destination, while also carrying out a normal delivery. In effect, the current address is made into a "parent" that has two children – one that is delivered as specified by this router, and a clone that goes on to be routed further. For this reason, **unseen** may not be combined with the **one_time** option in a *redirect* router.

Warning: Header lines added to the address (or specified for removal) by this router or by previous routers affect the "unseen" copy of the message only. The clone that continues to be processed by further routers starts with no added headers and none specified for removal. For a **redirect** router, if a generated address is the same as the incoming address, this can lead to duplicate addresses with different header modifications. Exim does not do duplicate deliveries (except, in certain circum-

stances, to pipes -- see section 22.7), but it is undefined which of the duplicates is discarded, so this ambiguous situation should be avoided. The **repeat_use** option of the **redirect** router may be of help.

Unlike the handling of header modifications, any data that was set by the **address_data** option in the current or previous routers *is* passed on to subsequent routers.

user	Use: *routers*	Type: *string*†	Default: *see below*

When a router queues an address for a transport, and the transport does not specify a user, the user given here is used when running the delivery process. The user may be specified numerically or by name. If expansion fails, the error is logged and delivery is deferred. This user is also used by the *redirect* router when running a filter file. The default is unset, except when **check_local_user** is set. In this case, the default is taken from the password information. If the user is specified as a name, and **group** is not set, the group associated with the user is used. See also **initgroups** and **group** and the discussion in chapter 23.

verify	Use: *routers*‡	Type: *boolean*	Default: *true*

Setting this option has the effect of setting **verify_sender** and **verify_recipient** to the same value.

verify_only	Use: *routers*‡	Type: *boolean*	Default: *false*

If this option is set, the router 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. It can be further restricted to verifying only senders or recipients by means of **verify_sender** and **verify_recipient**.

Warning: When the router is being run to verify addresses for an incoming SMTP message, Exim is not running as root, but under its own uid. If the router accesses any files, you need to make sure that they are accessible to the Exim user or group.

verify_recipient	Use: *routers*‡	Type: *boolean*	Default: *true*

If this option is false, the router is skipped when verifying recipient addresses or testing recipient verification using **-bv**. See section 3.12 for a list of the order in which preconditions are evaluated.

verify_sender	Use: *routers*‡	Type: *boolean*	Default: *true*

If this option is false, the router is skipped when verifying sender addresses or testing sender verifi¤ cation using **-bvs**. See section 3.12 for a list of the order in which preconditions are evaluated.

16. The accept router

The *accept* router has no private options of its own. Unless it is being used purely for verification (see **verify_only**) a transport is required to be defined by the generic **transport** option. If the preconditions that are specified by generic options are met, the router accepts the address and queues it for the given transport. The most common use of this router is for setting up deliveries to local mailboxes. For example:

```
localusers:
  driver = accept
  domains = mydomain.example
  check_local_user
  transport = local_delivery
```

The **domains** condition in this example checks the domain of the address, and **check_local_user** checks that the local part is the login of a local user. When both preconditions are met, the *accept* router runs, and queues the address for the *local_delivery* transport.

17. The dnslookup router

The *dnslookup* router looks up the hosts that handle mail for the recipient's domain in the DNS. A transport must always be set for this router, unless **verify_only** is set.

If SRV support is configured (see **check_srv** below), Exim first searches for SRV records. If none are found, or if SRV support is not configured, MX records are looked up. If no MX records exist, address records are sought. However, **mx_domains** can be set to disable the direct use of address records.

MX records of equal priority are sorted by Exim into a random order. Exim then looks for address records for the host names obtained from MX or SRV records. When a host has more than one IP address, they are sorted into a random order, except that IPv6 addresses are always sorted before IPv4 addresses. If all the IP addresses found are discarded by a setting of the **ignore_target_hosts** generic option, the router declines.

Unless they have the highest priority (lowest MX value), MX records that point to the local host, or to any host name that matches **hosts_treat_as_local**, are discarded, together with any other MX records of equal or lower priority.

If the host pointed to by the highest priority MX record, or looked up as an address record, is the local host, or matches **hosts_treat_as_local**, what happens is controlled by the generic **self** option.

17.1 Problems with DNS lookups

There have been problems with DNS servers when SRV records are looked up. Some mis-behaving servers return a DNS error or timeout when a non-existent SRV record is sought. Similar problems have in the past been reported for MX records. The global **dns_again_means_nonexist** option can help with this problem, but it is heavy-handed because it is a global option.

For this reason, there are two options, **srv_fail_domains** and **mx_fail_domains**, that control what happens when a DNS lookup in a *dnslookup* router results in a DNS failure or a "try again" response. If an attempt to look up an SRV or MX record causes one of these results, and the domain matches the relevant list, Exim behaves as if the DNS had responded "no such record". In the case of an SRV lookup, this means that the router proceeds to look for MX records; in the case of an MX lookup, it proceeds to look for A or AAAA records, unless the domain matches **mx_domains**, in which case routing fails.

17.2 Private options for dnslookup

The private options for the *dnslookup* router are as follows:

check_secondary_mx	Use: *dnslookup*	Type: *boolean*	Default: *false*

If this option is set, the router declines unless the local host is found in (and removed from) 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 to other domains. The way in which Exim decides whether a host is the local host is described in section 13.8.

check_srv	Use: *dnslookup*	Type: *string*†	Default: *unset*

The *dnslookup* router supports the use of SRV records (see RFC 2782) in addition to MX and address records. The support is disabled by default. To enable SRV support, set the **check_srv** option to the name of the service required. For example,

```
check_srv = smtp
```

looks for SRV records that refer to the normal smtp service. The option is expanded, so the service name can vary from message to message or address to address. This might be helpful if SRV records

are being used for a submission service. If the expansion is forced to fail, the **check_srv** option is ignored, and the router proceeds to look for MX records in the normal way.

When the expansion succeeds, the router searches first for SRV records for the given service (it assumes TCP protocol). A single SRV record with a host name that consists of just a single dot indicates "no such service for this domain"; if this is encountered, the router declines. If other kinds of SRV record are found, they are used to construct a host list for delivery according to the rules of RFC 2782. MX records are not sought in this case.

When no SRV records are found, MX records (and address records) are sought in the traditional way. In other words, SRV records take precedence over MX records, just as MX records take precedence over address records. Note that this behaviour is not sanctioned by RFC 2782, though a previous draft RFC defined it. It is apparently believed that MX records are sufficient for email and that SRV records should not be used for this purpose. However, SRV records have an additional "weight" feature which some people might find useful when trying to split an SMTP load between hosts of different power.

See section 17.1 above for a discussion of Exim's behaviour when there is a DNS lookup error.

mx_domains	Use: *dnslookup*	Type: *domain list*†	Default: *unset*

A domain that matches **mx_domains** is required to have either an MX or an SRV record in order to be recognized. (The name of this option could be improved.) For example, if all the mail hosts in *fict.example* are known to have MX records, except for those in *discworld.fict.example*, you could use this setting:

```
mx_domains = ! *.discworld.fict.example : *.fict.example
```

This specifies that messages addressed to a domain that matches the list but has no MX record should be bounced immediately instead of being routed using the address record.

mx_fail_domains	Use: *dnslookup*	Type: *domain list*†	Default: *unset*

If the DNS lookup for MX records for one of the domains in this list causes a DNS lookup error, Exim behaves as if no MX records were found. See section 17.1 for more discussion.

qualify_single	Use: *dnslookup*	Type: *boolean*	Default: *true*

When this option is true, the resolver option RES_DEFNAMES is set for DNS lookups. Typically, but not standardly, this causes the resolver to qualify single-component names with the default domain. For example, on a machine called *dictionary.ref.example*, the domain *thesaurus* would be changed to *thesaurus.ref.example* inside the resolver. For details of what your resolver actually does, consult your man pages for *resolver* and *resolv.conf*.

rewrite_headers	Use: *dnslookup*	Type: *boolean*	Default: *true*

If the domain name in the address that is being processed is not fully qualified, it may be expanded to its full form by a DNS lookup. For example, if an address is specified as *dormouse@teaparty*, the domain might be expanded to *teaparty.wonderland.fict.example*. Domain expansion can also occur as a result of setting the **widen_domains** option. If **rewrite_headers** is true, all occurrences of the abbreviated domain name in any *Bcc:*, *Cc:*, *From:*, *Reply-to:*, *Sender:*, and *To:* header lines of the message are rewritten with the full domain name.

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.

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 was present or not. However, some name servers have recently been seen to return the

wildcard entry. If the name returned by a DNS lookup begins with an asterisk, it is not used for header rewriting.

same_domain_copy_routing	Use: *dnslookup*	Type: *boolean*	Default: *false*

Addresses with the same domain are normally routed by the *dnslookup* router to the same list of hosts. However, this cannot be presumed, because the router options and preconditions may refer to the local part of the address. By default, therefore, Exim routes each address in a message independently. DNS servers run caches, so repeated DNS lookups are not normally expensive, and in any case, personal messages rarely have more than a few recipients.

If you are running mailing lists with large numbers of subscribers at the same domain, and you are using a *dnslookup* router which is independent of the local part, you can set **same_domain_copy_routing** to bypass repeated DNS lookups for identical domains in one message. In this case, when *dnslookup* routes an address to a remote transport, any other unrouted addresses in the message that have the same domain are automatically given the same routing without processing them independently, provided the following conditions are met:

- No router that processed the address specified **headers_add** or **headers_remove**.
- The router did not change the address in any way, for example, by "widening" the domain.

search_parents	Use: *dnslookup*	Type: *boolean*	Default: *false*

When this option is true, the resolver option RES_DNSRCH is set for DNS lookups. This is different from the **qualify_single** option in that it applies to domains containing dots. Typically, but not standardly, it causes the resolver to search for the name in the current domain and in parent domains. For example, on a machine in the *fict.example* domain, if looking up *teaparty.wonderland* failed, the resolver would try *teaparty.wonderland.fict.example*. For details of what your resolver actually does, consult your man pages for *resolver* and *resolv.conf*.

Setting this option true can cause problems in domains that have a wildcard MX record, because any domain that does not have its own MX record matches the local wildcard.

srv_fail_domains	Use: *dnslookup*	Type: *domain list*†	Default: *unset*

If the DNS lookup for SRV records for one of the domains in this list causes a DNS lookup error, Exim behaves as if no SRV records were found. See section 17.1 for more discussion.

widen_domains	Use: *dnslookup*	Type: *string list*	Default: *unset*

If a DNS 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. For example, if

```
widen_domains = fict.example:ref.example
```

is set and a lookup of *klingon.dictionary* fails, *klingon.dictionary.fict.example* is looked up, and if this fails, *klingon.dictionary.ref.example* is tried. Note that the **qualify_single** and **search_parents** options can cause some widening to be undertaken inside the DNS resolver. **widen_domains** is not applied to sender addresses when verifying, unless **rewrite_headers** is false (not the default).

17.3 Effect of qualify_single and search_parents

When a domain from an envelope recipient is changed by the resolver as a result of the **qualify_single** or **search_parents** options, Exim rewrites the corresponding address in the message's header lines unless **rewrite_headers** is set false. Exim then re-routes the address, using the full domain.

These two options affect only the DNS lookup that takes place inside the router for the domain of the address that is being routed. They do not affect lookups such as that implied by

```
domains = @mx_any
```

that may happen while processing a router precondition before the router is entered. No widening ever takes place for these lookups.

18. The ipliteral router

This router has no private options. Unless it is being used purely for verification (see **verify_only**) a transport is required to be defined by the generic **transport** option. The router accepts the address if its domain part takes the form of an RFC 2822 domain literal. For example, the *ipliteral* router handles the address

```
root@[192.168.1.1]
```

by setting up delivery to the host with that IP address. IPv4 domain literals consist of an IPv4 address enclosed in square brackets. IPv6 domain literals are similar, but the address is preceded by `ipv6:`. For example:

```
postmaster@[ipv6:fe80::a00:20ff:fe86:a061.5678]
```

Exim allows `ipv4:` before IPv4 addresses, for consistency, and on the grounds that sooner or later somebody will try it.

If the IP address matches something in **ignore_target_hosts**, the router declines. If an IP literal turns out to refer to the local host, the generic **self** option determines what happens.

The RFCs require support for domain literals; however, their use is controversial in today's Internet. If you want to use this router, you must also set the main configuration option **allow_domain_literals**. Otherwise, Exim will not recognize the domain literal syntax in addresses.

19. The iplookup router

The *iplookup* router was written to fulfil a specific requirement in Cambridge University (which in fact no longer exists). For this reason, it is not included in the binary of Exim by default. If you want to include it, you must set

```
ROUTER_IPLOOKUP=yes
```

in your *Local/Makefile* configuration file.

The *iplookup* router routes an address by sending it over a TCP or UDP connection to one or more specific hosts. The host can then return the same or a different address – in effect rewriting the recipient address in the message's envelope. The new address is then passed on to subsequent routers. If this process fails, the address can be passed on to other routers, or delivery can be deferred. Since *iplookup* is just a rewriting router, a transport must not be specified for it.

hosts	Use: *iplookup*	Type: *string*	Default: *unset*

This option must be supplied. Its value is a colon-separated list of host names. The hosts are looked up using *gethostbyname()* (or *getipnodebyname()* when available) and are tried in order until one responds to the query. If none respond, what happens is controlled by **optional**.

optional	Use: *iplookup*	Type: *boolean*	Default: *false*

If **optional** is true, if no response is obtained from any host, the address is passed to the next router, overriding **no_more**. If **optional** is false, delivery to the address is deferred.

port	Use: *iplookup*	Type: *integer*	Default: *0*

This option must be supplied. It specifies the port number for the TCP or UDP call.

protocol	Use: *iplookup*	Type: *string*	Default: *udp*

This option can be set to "udp" or "tcp" to specify which of the two protocols is to be used.

query	Use: *iplookup*	Type: *string*†	Default: *see below*

This defines the content of the query that is sent to the remote hosts. The default value is:

```
$local_part@$domain $local_part@$domain
```

The repetition serves as a way of checking that a response is to the correct query in the default case (see **response_pattern** below).

reroute	Use: *iplookup*	Type: *string*†	Default: *unset*

If this option is not set, the rerouted address is precisely the byte string returned by the remote host, up to the first white space, if any. If set, the string is expanded to form the rerouted address. It can include parts matched in the response by **response_pattern** by means of numeric variables such as *$1*, *$2*, etc. The variable *$0* refers to the entire input string, whether or not a pattern is in use. In all cases, the rerouted address must end up in the form *local_part@domain*.

response_pattern	Use: *iplookup*	Type: *string*	Default: *unset*

This option can be set to a regular expression that is applied to the string returned from the remote host. If the pattern does not match the response, the router declines. If **response_pattern** is not set, no checking of the response is done, unless the query was defaulted, in which case there is a check that the text returned after the first white space is the original address. This checks that the answer that has been received is in response to the correct question. For example, if the response is just a new domain, the following could be used:

```
response_pattern = ^([^@]+)$
reroute = $local_part@$1
```

timeout	Use: *iplookup*	Type: *time*	Default: *5s*

This specifies the amount of time to wait for a response from the remote machine. The same timeout is used for the *connect()* function for a TCP call. It does not apply to UDP.

20. The manualroute router

The *manualroute* router is so-called because it provides a way of manually routing an address according to its domain. It is mainly used when you want to route addresses to remote hosts according to your own rules, bypassing the normal DNS routing that looks up MX records. However, *manualroute* can also route to local transports, a facility that may be useful if you want to save messages for dial-in hosts in local files.

The *manualroute* router compares a list of domain patterns with the domain it is trying to route. If there is no match, the router declines. Each pattern has associated with it a list of hosts and some other optional data, which may include a transport. The combination of a pattern and its data is called a "routing rule". For patterns that do not have an associated transport, the generic **transport** option must specify a transport, unless the router is being used purely for verification (see **verify_only**).

In the case of verification, matching the domain pattern is sufficient for the router to accept the address. When actually routing an address for delivery, an address that matches a domain pattern is queued for the associated transport. If the transport is not a local one, a host list must be associated with the pattern; IP addresses are looked up for the hosts, and these are passed to the transport along with the mail address. For local transports, a host list is optional. If it is present, it is passed in *$host* as a single text string.

The list of routing rules can be provided as an inline string in **route_list**, or the data can be obtained by looking up the domain in a file or database by setting **route_data**. Only one of these settings may appear in any one instance of *manualroute*. The format of routing rules is described below, following the list of private options.

20.1 Private options for manualroute

The private options for the *manualroute* router are as follows:

host_all_ignored	Use: *manualroute*	Type: *string*	Default: *defer*

See **host_find_failed**.

host_find_failed	Use: *manualroute*	Type: *string*	Default: *freeze*

This option controls what happens when *manualroute* tries to find an IP address for a host, and the host does not exist. The option can be set to one of the following values:

```
decline
defer
fail
freeze
ignore
pass
```

The default ("freeze") assumes that this state is a serious configuration error. The difference between "pass" and "decline" is that the former forces the address to be passed to the next router (or the router defined by **pass_router**), overriding **no_more**, whereas the latter passes the address to the next router only if **more** is true.

The value "ignore" causes Exim to completely ignore a host whose IP address cannot be found. If all the hosts in the list are ignored, the behaviour is controlled by the **host_all_ignored** option. This takes the same values as **host_find_failed**, except that it cannot be set to "ignore".

The **host_find_failed** option applies only to a definite "does not exist" state; if a host lookup gets a temporary error, delivery is deferred unless the generic **pass_on_timeout** option is set.

hosts_randomize	Use: *manualroute*	Type: *boolean*	Default: *false*

If this option is set, the order of the items in a host list in a routing rule is randomized each time the list is used, unless an option in the routing rule overrides (see below). Randomizing the order of a host list can be used to do crude load sharing. However, if more than one mail address is routed by the same router to the same host list, the host lists are considered to be the same (even though they may be randomized into different orders) for the purpose of deciding whether to batch the deliveries into a single SMTP transaction.

When **hosts_randomize** is true, a host list may be split into groups whose order is separately randomized. This makes it possible to set up MX-like behaviour. The boundaries between groups are indicated by an item that is just + in the host list. For example:

```
route_list = * host1:host2:host3:+:host4:host5
```

The order of the first three hosts and the order of the last two hosts is randomized for each use, but the first three always end up before the last two. If **hosts_randomize** is not set, a + item in the list is ignored. If a randomized host list is passed to an *smtp* transport that also has **hosts_randomize set**, the list is not re-randomized.

route_data	Use: *manualroute*	Type: *string*†	Default: *unset*

If this option is set, it must expand to yield the data part of a routing rule. Typically, the expansion string includes a lookup based on the domain. For example:

```
route_data = ${lookup{$domain}dbm{/etc/routes}}
```

If the expansion is forced to fail, or the result is an empty string, the router declines. Other kinds of expansion failure cause delivery to be deferred.

route_list	Use: *manualroute*	Type: *string list*	Default: *unset*

This string is a list of routing rules, in the form defined below. Note that, unlike most string lists, the items are separated by semicolons. This is so that they may contain colon-separated host lists.

same_domain_copy_routing	Use: *manualroute*	Type: *boolean*	Default: *false*

Addresses with the same domain are normally routed by the *manualroute* router to the same list of hosts. However, this cannot be presumed, because the router options and preconditions may refer to the local part of the address. By default, therefore, Exim routes each address in a message independently. DNS servers run caches, so repeated DNS lookups are not normally expensive, and in any case, personal messages rarely have more than a few recipients.

If you are running mailing lists with large numbers of subscribers at the same domain, and you are using a *manualroute* router which is independent of the local part, you can set **same_domain_copy_routing** to bypass repeated DNS lookups for identical domains in one message. In this case, when *manualroute* routes an address to a remote transport, any other unrouted addresses in the message that have the same domain are automatically given the same routing without processing them independently. However, this is only done if **headers_add** and **headers_remove** are unset.

20.2 Routing rules in route_list

The value of **route_list** is a string consisting of a sequence of routing rules, separated by semicolons. If a semicolon is needed in a rule, it can be entered as two semicolons. Alternatively, the list separator can be changed as described (for colon-separated lists) in section 6.19. Empty rules are ignored. The format of each rule is

<*domain pattern*> <*list of hosts*> <*options*>

The following example contains two rules, each with a simple domain pattern and no options:

```
route_list = \
  dict.ref.example  mail-1.ref.example:mail-2.ref.example ; \
  thes.ref.example  mail-3.ref.example:mail-4.ref.example
```

The three parts of a rule are separated by white space. The pattern and the list of hosts can be enclosed in quotes if necessary, and if they are, the usual quoting rules apply. Each rule in a **route_list** must start with a single domain pattern, which is the only mandatory item in the rule. The pattern is in the same format as one item in a domain list (see section 10.8), except that it may not be the name of an interpolated file. That is, it may be wildcarded, or a regular expression, or a file or database lookup (with semicolons doubled, because of the use of semicolon as a separator in a **route_list**).

The rules in **route_list** are searched in order until one of the patterns matches the domain that is being routed. The list of hosts and then options are then used as described below. If there is no match, the router declines. When **route_list** is set, **route_data** must not be set.

20.3 Routing rules in route_data

The use of **route_list** is convenient when there are only a small number of routing rules. For larger numbers, it is easier to use a file or database to hold the routing information, and use the **route_data** option instead. The value of **route_data** is a list of hosts, followed by (optional) options. Most commonly, **route_data** is set as a string that contains an expansion lookup. For example, suppose we place two routing rules in a file like this:

```
dict.ref.example:  mail-1.ref.example:mail-2.ref.example
thes.ref.example:  mail-3.ref.example:mail-4.ref.example
```

This data can be accessed by setting

```
route_data = ${lookup{$domain}lsearch{/the/file/name}}
```

Failure of the lookup results in an empty string, causing the router to decline. However, you do not have to use a lookup in **route_data**. The only requirement is that the result of expanding the string is a list of hosts, possibly followed by options, separated by white space. The list of hosts must be enclosed in quotes if it contains white space.

20.4 Format of the list of hosts

A list of hosts, whether obtained via **route_data** or **route_list**, is always separately expanded before use. If the expansion fails, the router declines. The result of the expansion must be a colon-separated list of names and/or IP addresses, optionally also including ports. The format of each item in the list is described in the next section. The list separator can be changed as described in section 6.19.

If the list of hosts was obtained from a **route_list** item, the following variables are set during its expansion:

- If the domain was matched against a regular expression, the numeric variables *$1*, *$2*, etc. may be set. For example:

  ```
  route_list = ^domain(\d+)   host-$1.text.example
  ```

- *$0* is always set to the entire domain.
- *$1* is also set when partial matching is done in a file lookup.
- If the pattern that matched the domain was a lookup item, the data that was looked up is available in the expansion variable *$value*. For example:

  ```
  route_list = lsearch;;/some/file.routes  $value
  ```

Note the doubling of the semicolon in the pattern that is necessary because semicolon is the default route list separator.

20.5 Format of one host item

Each item in the list of hosts is either a host name or an IP address, optionally with an attached port number. When no port is given, an IP address is not enclosed in brackets. When a port is specified, it overrides the port specification on the transport. The port is separated from the name or address by a colon. This leads to some complications:

- Because colon is the default separator for the list of hosts, either the colon that specifies a port must be doubled, or the list separator must be changed. The following two examples have the same effect:

```
route_list = * "host1.tld::1225 : host2.tld::1226"
route_list = * "<+ host1.tld:1225 + host2.tld:1226"
```

- When IPv6 addresses are involved, it gets worse, because they contain colons of their own. To make this case easier, it is permitted to enclose an IP address (either v4 or v6) in square brackets if a port number follows. For example:

```
route_list = * "</ [10.1.1.1]:1225 / [::1]:1226"
```

20.6 How the list of hosts is used

When an address is routed to an *smtp* transport by *manualroute*, each of the hosts is tried, in the order specified, when carrying out the SMTP delivery. However, the order can be changed by setting the **hosts_randomize** option, either on the router (see section 20.1 above), or on the transport.

Hosts may be listed by name or by IP address. An unadorned name in the list of hosts is interpreted as a host name. A name that is followed by `/MX` is interpreted as an indirection to a sublist of hosts obtained by looking up MX records in the DNS. For example:

```
route_list = *  x.y.z:p.q.r/MX:e.f.g
```

If this feature is used with a port specifier, the port must come last. For example:

```
route_list = *  dom1.tld/mx::1225
```

If the **hosts_randomize** option is set, the order of the items in the list is randomized before any lookups are done. Exim then scans the list; for any name that is not followed by `/MX` it looks up an IP address. If this turns out to be an interface on the local host and the item is not the first in the list, Exim discards it and any subsequent items. If it is the first item, what happens is controlled by the **self** option of the router.

A name on the list that is followed by `/MX` is replaced with the list of hosts obtained by looking up MX records for the name. This is always a DNS lookup; the **bydns** and **byname** options (see section 20.7 below) are not relevant here. The order of these hosts is determined by the preference values in the MX records, according to the usual rules. Because randomizing happens before the MX lookup, it does not affect the order that is defined by MX preferences.

If the local host is present in the sublist obtained from MX records, but is not the most preferred host in that list, it and any equally or less preferred hosts are removed before the sublist is inserted into the main list.

If the local host is the most preferred host in the MX list, what happens depends on where in the original list of hosts the `/MX` item appears. If it is not the first item (that is, there are previous hosts in the main list), Exim discards this name and any subsequent items in the main list.

If the MX item is first in the list of hosts, and the local host is the most preferred host, what happens is controlled by the **self** option of the router.

DNS failures when lookup up the MX records are treated in the same way as DNS failures when looking up IP addresses: **pass_on_timeout** and **host_find_failed** are used when relevant.

The generic **ignore_target_hosts** option applies to all hosts in the list, whether obtained from an MX lookup or not.

20.7 How the options are used

The options are a sequence of words; in practice no more than three are ever present. One of the words can be the name of a transport; this overrides the **transport** option on the router for this particular routing rule only. The other words (if present) control randomization of the list of hosts on a per-rule basis, and how the IP addresses of the hosts are to be found when routing to a remote transport. These options are as follows:

- **randomize**: randomize the order of the hosts in this list, overriding the setting of **hosts_randomize** for this routing rule only.
- **no_randomize**: do not randomize the order of the hosts in this list, overriding the setting of **hosts_randomize** for this routing rule only.
- **byname**: use *getipnodebyname()* (*gethostbyname()* on older systems) to find IP addresses. This function may ultimately cause a DNS lookup, but it may also look in */etc/hosts* or other sources of information.
- **bydns**: look up address records for the hosts directly in the DNS; fail if no address records are found. If there is a temporary DNS error (such as a timeout), delivery is deferred.

For example:

```
route_list = domain1  host1:host2:host3  randomize bydns;\
             domain2  host4:host5
```

If neither **byname** nor **bydns** is given, Exim behaves as follows: First, a DNS lookup is done. If this yields anything other than HOST_NOT_FOUND, that result is used. Otherwise, Exim goes on to try a call to *getipnodebyname()* or *gethostbyname()*, and the result of the lookup is the result of that call.

Warning: It has been discovered that on some systems, if a DNS lookup called via *getipnodebyname()* times out, HOST_NOT_FOUND is returned instead of TRY_AGAIN. That is why the default action is to try a DNS lookup first. Only if that gives a definite "no such host" is the local function called.

If no IP address for a host can be found, what happens is controlled by the **host_find_failed** option.

When an address is routed to a local transport, IP addresses are not looked up. The host list is passed to the transport in the *$host* variable.

20.8 Manualroute examples

In some of the examples that follow, the presence of the **remote_smtp** transport, as defined in the default configuration file, is assumed:

- The *manualroute* router can be used to forward all external mail to a *smart host*. If you have set up, in the main part of the configuration, a named domain list that contains your local domains, for example:

  ```
  domainlist local_domains = my.domain.example
  ```

  You can arrange for all other domains to be routed to a smart host by making your first router something like this:

  ```
  smart_route:
    driver = manualroute
    domains = !+local_domains
    transport = remote_smtp
    route_list = * smarthost.ref.example
  ```

  This causes all non-local addresses to be sent to the single host *smarthost.ref.example*. If a colon-separated list of smart hosts is given, they are tried in order (but you can use **hosts_randomize** to vary the order each time). Another way of configuring the same thing is this:

  ```
  smart_route:
    driver = manualroute
  ```

```
  transport = remote_smtp
  route_list = !+local_domains  smarthost.ref.example
```

There is no difference in behaviour between these two routers as they stand. However, they behave differently if **no_more** is added to them. In the first example, the router is skipped if the domain does not match the **domains** precondition; the following router is always tried. If the router runs, it always matches the domain and so can never decline. Therefore, **no_more** would have no effect. In the second case, the router is never skipped; it always runs. However, if it doesn't match the domain, it declines. In this case **no_more** would prevent subsequent routers from running.

- A *mail hub* is a host which receives mail for a number of domains via MX records in the DNS and delivers it via 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 *manualroute* router is usually used on a mail hub to route incoming messages to the correct hosts. For a small number of domains, the routing can be inline, using the **route_list** option, but for a larger number a file or database lookup is easier to manage.

  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 quite simple. For example:

```
hub_route:
  driver = manualroute
  transport = remote_smtp
  route_list = *.rhodes.tvs.example  $domain
```

  This configuration routes domains that match `*.rhodes.tvs.example` to hosts whose names are the same as the mail domains. A similar approach can be taken if the host name can be obtained from the domain name by a string manipulation that the expansion facilities can handle. Otherwise, a lookup based on the domain can be used to find the host:

```
through_firewall:
  driver = manualroute
  transport = remote_smtp
  route_data = ${lookup {$domain} cdb {/internal/host/routes}}
```

  The result of the lookup must be the name or IP address of the host (or hosts) to which the address is to be routed. If the lookup fails, the route data is empty, causing the router to decline. The address then passes to the next router.

- You can use *manualroute* to deliver messages to pipes or files in batched SMTP format for onward transportation by some other means. This is one way of storing mail for a dial-up host when it is not connected. The route list entry can be as simple as a single domain name in a configuration like this:

```
save_in_file:
  driver = manualroute
  transport = batchsmtp_appendfile
  route_list = saved.domain.example
```

  though often a pattern is used to pick up more than one domain. If there are several domains or groups of domains with different transport requirements, different transports can be listed in the routing information:

```
save_in_file:
  driver = manualroute
  route_list = \
    *.saved.domain1.example  $domain  batch_appendfile; \
    *.saved.domain2.example  \
      ${lookup{$domain}dbm{/domain2/hosts}{$value}fail} \
      batch_pipe
```

  The first of these just passes the domain in the *$host* variable, which doesn't achieve much (since it is also in *$domain*), but the second does a file lookup to find a value to pass, causing the router to decline to handle the address if the lookup fails.

- Routing mail directly to UUCP software is a specific case of the use of *manualroute* in a gateway to another mail environment. This is an example of one way it can be done:

```
# Transport
uucp:
  driver = pipe
  user = nobody
  command = /usr/local/bin/uux -r - \
    ${substr_-5:$host}!rmail ${local_part}
  return_fail_output = true

# Router
uucphost:
  transport = uucp
  driver = manualroute
  route_data = \
    ${lookup{$domain}lsearch{/usr/local/exim/uucphosts}}
```

  The file */usr/local/exim/uucphosts* contains entries like

```
darksite.ethereal.example:           darksite.UUCP
```

  It can be set up more simply without adding and removing ".UUCP" but this way makes clear the distinction between the domain name *darksite.ethereal.example* and the UUCP host name *darksite*.

21. The queryprogram router

The *queryprogram* router routes an address by running an external command and acting on its output. This 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 use the precondition options (**domains**, **local_parts**, etc) to skip this router for most addresses, it could sensibly be used in special cases, even on a busy host. There are the following private options:

command	Use: *queryprogram*	Type: *string*†	Default: *unset*

This option must be set. It specifies the command that is to be run. The command is split up into a command name and arguments, and then each is expanded separately (exactly as for a *pipe* transport, described in chapter 29).

command_group	Use: *queryprogram*	Type: *string*	Default: *unset*

This option specifies a gid to be set when running the command while routing an address for deliver. It must be set if **command_user** specifies a numerical uid. If it begins with a digit, it is interpreted as the numerical value of the gid. Otherwise it is looked up using *getgrnam()*.

command_user	Use: *queryprogram*	Type: *string*	Default: *unset*

This option must be set. It specifies the uid which is set when running the command while routing an address for delivery. If the value begins with a digit, it is interpreted as the numerical value of the uid. Otherwise, it is looked up using *getpwnam()* to obtain a value for the uid and, if **command_group** is not set, a value for the gid also.

Warning: Changing uid and gid is possible only when Exim is running as root, which it does during a normal delivery in a conventional configuration. However, when an address is being verified during message reception, Exim is usually running as the Exim user, not as root. If the *queryprogram* router is called from a non-root process, Exim cannot change uid or gid before running the command. In this circumstance the command runs under the current uid and gid.

current_directory	Use: *queryprogram*	Type: *string*	Default: */*

This option specifies an absolute path which is made the current directory before running the command.

timeout	Use: *queryprogram*	Type: *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.

The standard output of the command is connected to a pipe, which is read when the command terminates. It should consist of a single line of output, containing up to five fields, separated by white space. The maximum length of the line is 1023 characters. Longer lines are silently truncated. The first field is one of the following words (case-insensitive):

- *Accept*: routing succeeded; the remaining fields specify what to do (see below).
- *Decline*: the router declines; pass the address to the next router, unless **no_more** is set.
- *Fail*: routing failed; do not pass the address to any more routers. Any subsequent text on the line is an error message. If the router is run as part of address verification during an incoming SMTP message, the message is included in the SMTP response.

- *Defer*: routing could not be completed at this time; try again later. Any subsequent text on the line is an error message which is logged. It is not included in any SMTP response.

- *Freeze*: the same as *defer*, except that the message is frozen.

- *Pass*: pass the address to the next router (or the router specified by **pass_router**), overriding **no_more**.

- *Redirect*: the message is redirected. The remainder of the line is a list of new addresses, which are routed independently, starting with the first router, or the router specified by **redirect_router**, if set.

When the first word is *accept*, the remainder of the line consists of a number of keyed data values, as follows (split into two lines here, to fit on the page):

```
ACCEPT TRANSPORT=<transport> HOSTS=<list of hosts>
LOOKUP=byname|bydns DATA=<text>
```

The data items can be given in any order, and all are optional. If no transport is included, the transport specified by the generic **transport** option is used. The list of hosts and the lookup type are needed only if the transport is an *smtp* transport that does not itself supply a list of hosts.

The format of the list of hosts is the same as for the *manualroute* router. As well as host names and IP addresses with optional port numbers, as described in section 20.5, it may contain names followed by `/MX` to specify sublists of hosts that are obtained by looking up MX records (see section 20.6).

If the lookup type is not specified, Exim behaves as follows when trying to find an IP address for each host: First, a DNS lookup is done. If this yields anything other than HOST_NOT_FOUND, that result is used. Otherwise, Exim goes on to try a call to *getipnodebyname()* or *gethostbyname()*, and the result of the lookup is the result of that call.

If the DATA field is set, its value is placed in the *$address_data* variable. For example, this return line

```
accept hosts=x1.y.example:x2.y.example data="rule1"
```

routes the address to the default transport, passing a list of two hosts. When the transport runs, the string “rule1” is in *$address_data*.

22. The redirect router

The *redirect* router handles several kinds of address redirection. Its most common uses are for resolving local part aliases from a central alias file (usually called */etc/aliases*) and for handling users' personal *.forward* files, but it has many other potential uses. The incoming address can be redirected in several different ways:

- It can be replaced by one or more new addresses which are themselves routed independently.
- It can be routed to be delivered to a given file or directory.
- It can be routed to be delivered to a specified pipe command.
- It can cause an automatic reply to be generated.
- It can be forced to fail, optionally with a custom error message.
- It can be temporarily deferred, optionally with a custom message.
- It can be discarded.

The generic **transport** option must not be set for *redirect* routers. However, there are some private options which define transports for delivery to files and pipes, and for generating autoreplies. See the **file_transport**, **pipe_transport** and **reply_transport** descriptions below.

22.1 Redirection data

The router operates by interpreting a text string which it obtains either by expanding the contents of the **data** option, or by reading the entire contents of a file whose name is given in the **file** option. These two options are mutually exclusive. The first is commonly used for handling system aliases, in a configuration like this:

```
system_aliases:
  driver = redirect
  data = ${lookup{$local_part}lsearch{/etc/aliases}}
```

If the lookup fails, the expanded string in this example is empty. When the expansion of **data** results in an empty string, the router declines. A forced expansion failure also causes the router to decline; other expansion failures cause delivery to be deferred.

A configuration using **file** is commonly used for handling users' *.forward* files, like this:

```
userforward:
  driver = redirect
  check_local_user
  file = $home/.forward
  no_verify
```

If the file does not exist, or causes no action to be taken (for example, it is empty or consists only of comments), the router declines. **Warning**: This is not the case when the file contains syntactically valid items that happen to yield empty addresses, for example, items containing only RFC 2822 address comments.

22.2 Forward files and address verification

It is usual to set **no_verify** on *redirect* routers which handle users' *.forward* files, as in the example above. There are two reasons for this:

- When Exim is receiving an incoming SMTP message from a remote host, it is running under the Exim uid, not as root. Exim is unable to change uid to read the file as the user, and it may not be able to read it as the Exim user. So in practice the router may not be able to operate.
- However, even when the router can operate, the existence of a *.forward* file is unimportant when verifying an address. What should be checked is whether the local part is a valid user name or not. Cutting out the redirection processing saves some resources.

22.3 Interpreting redirection data

The contents of the data string, whether obtained from **data** or **file**, can be interpreted in two different ways:

- If the **allow_filter** option is set true, and the data begins with the text "#Exim filter" or "#Sieve filter", it is interpreted as a list of *filtering* instructions in the form of an Exim or Sieve filter file, respectively. Details of the syntax and semantics of filter files are described in a separate document entitled *Exim's interfaces to mail filtering*; this document is intended for use by end users.
- Otherwise, the data must be a comma-separated list of redirection items, as described in the next section.

When a message is redirected to a file (a "mail folder"), the file name given in a non-filter redirection list must always be an absolute path. A filter may generate a relative path – how this is handled depends on the transport's configuration. See section 26.1 for a discussion of this issue for the *appendfile* transport.

22.4 Items in a non-filter redirection list

When the redirection data is not an Exim or Sieve filter, for example, if it comes from a conventional alias or forward file, it consists of a list of addresses, file names, pipe commands, or certain special items (see section 22.6 below). The special items can be individually enabled or disabled by means of options whose names begin with **allow_** or **forbid_**, depending on their default values. The items in the list are separated by commas or newlines. If a comma is required in an item, the entire item must be enclosed in double quotes.

Lines starting with a # character are comments, and are ignored, and # may also appear following a comma, in which case everything between the # and the next newline character is 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.

Warning: If you use an Exim expansion to construct a redirection address, and the expansion contains a reference to *$local_part*, you should make use of the **quote_local_part** expansion operator, in case the local part contains special characters. For example, to redirect all mail for the domain *obsolete.example*, retaining the existing local part, you could use this setting:

```
data = ${quote_local_part:$local_part}@newdomain.example
```

22.5 Redirecting to a local mailbox

A redirection item may safely be the same as the address currently under consideration. This does not cause a routing loop, because a router is automatically skipped if any ancestor of the address that is being processed is the same as the current address and was processed by the current router. Such an address is therefore passed to the following routers, so it is handled as if there were no redirection. When making this loop-avoidance test, the complete local part, including any prefix or suffix, is used.

Specifying the same local part without a domain is a common usage in personal filter files when the user wants to have messages delivered to the local mailbox and also forwarded elsewhere. For example, the user whose login is *cleo* might have a *.forward* file containing this:

```
cleo, cleopatra@egypt.example
```

For compatibility with other MTAs, such unqualified local parts may be preceded by "\", but this is not a requirement for loop prevention. However, it does make a difference if more than one domain is being handled synonymously.

If an item begins with "\" and the rest of the item parses as a valid RFC 2822 address that does not include a domain, the item is qualified using the domain of the incoming address. In the absence of a leading "\", unqualified addresses are qualified using the value in **qualify_recipient**, but you can force the incoming domain to be used by setting **qualify_preserve_domain**.

Care must be taken if there are alias names for local users. Consider an MTA handling a single local domain where the system alias file contains:

```
Sam.Reman: spqr
```

Now suppose that Sam (whose login id is *spqr*) wants to save copies of messages in the local mailbox, and also forward copies elsewhere. He creates this forward file:

```
Sam.Reman, spqr@reme.elsewhere.example
```

With these settings, an incoming message addressed to *Sam.Reman* fails. The *redirect* router for system aliases does not process *Sam.Reman* the second time round, because it has previously routed it, and the following routers presumably cannot handle the alias. The forward file should really contain

```
spqr, spqr@reme.elsewhere.example
```

but because this is such a common error, the **check_ancestor** option (see below) exists to provide a way to get round it. This is normally set on a *redirect* router that is handling users' *.forward* files.

22.6 Special items in redirection lists

In addition to addresses, the following types of item may appear in redirection lists (that is, in non-filter redirection data):

- An item is treated as a pipe command if it begins with "|" and does not parse as a valid RFC 2822 address that includes a domain. A transport for running the command must be specified by the **pipe_transport** option. Normally, either the router or the transport specifies a user and a group under which to run the delivery. The default is to use the Exim user and group.

  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, for example:

  ```
  "|/some/command ready,steady,go"
  ```

  since items in redirection lists 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.

- An item is interpreted as a path name if it begins with "/" and does not parse as a valid RFC 2822 address that includes a domain. For example,

  ```
  /home/world/minbari
  ```

  is treated as a file name, but

  ```
  /s=molari/o=babylon/@x400gate.way
  ```

  is treated as an address. For a file name, a transport must be specified using the **file_transport** option. However, if the generated path name ends with a forward slash character, it is interpreted as a directory name rather than a file name, and **directory_transport** is used instead.

  Normally, either the router or the transport specifies a user and a group under which to run the delivery. The default is to use the Exim user and group.

  However, if a redirection item is the path */dev/null*, delivery to it is bypassed at a high level, and the log entry shows "**bypassed**" instead of a transport name. In this case the user and group are not used.

- If an item is of the form

  ```
  :include:<path name>
  ```

  a list of further items is taken from the given file and included at that point. **Note**: Such a file can not be a filter file; it is just an out-of-line addition to the list. The items in the included list are

separated by commas or newlines and are not subject to expansion. If this is the first item in an alias list in an *lsearch* file, a colon must be used to terminate the alias name. This example is incorrect:

```
list1    :include:/opt/lists/list1
```

It must be given as

```
list1:   :include:/opt/lists/list1
```

- Sometimes you want to throw away mail to a particular local part. Making the **data** option expand to an empty string does not work, because that causes the router to decline. Instead, the alias item *:blackhole:* can be used. It does what its name implies. No delivery is done, and no error message is generated. This has the same effect as specifing */dev/null* as a destination, but it can be independently disabled.

  Warning: If *:blackhole:* appears anywhere in a redirection list, no delivery is done for the original local part, even if other redirection items are present. If you are generating a multi-item list (for example, by reading a database) and need the ability to provide a no-op item, you must use */dev/null*.

- An attempt to deliver a particular address can be deferred or forced to fail by redirection items of the form

  ```
  :defer:
  :fail:
  ```

  respectively. When a redirection list contains such an item, it applies to the entire redirection; any other items in the list are ignored (*:blackhole:* is different). Any text following *:fail:* or *:defer:* is placed in the error text associated with the failure. For example, an alias file might contain:

  ```
  X.Employee:  :fail: Gone away, no forwarding address
  ```

  In the case of an address that is being verified from an ACL or as the subject of a VRFY command, the text is included in the SMTP error response by default. The text is not included in the response to an EXPN command. In non-SMTP cases the text is included in the error message that Exim generates.

  By default, Exim sends a 451 SMTP code for a *:defer:*, and 550 for *:fail:*. However, if the message starts with three digits followed by a space, optionally followed by an extended code of the form *n.n.n*, also followed by a space, and the very first digit is the same as the default error code, the code from the message is used instead. If the very first digit is incorrect, a panic error is logged, and the default code is used. You can suppress the use of the supplied code in a redirect router by setting the **forbid_smtp_code** option true. In this case, any SMTP code is quietly ignored.

  In an ACL, an explicitly provided message overrides the default, but the default message is available in the variable *$acl_verify_message* and can therefore be included in a custom message if this is desired.

  Normally the error text is the rest of the redirection list – a comma does not terminate it – but a newline does act as a terminator. 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 *:include:* files.

  During routing for message delivery (as opposed to verification), a redirection 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 the normal retry rules still apply.

- Sometimes it is useful to use a single-key search type with a default (see chapter 9) to look up aliases. However, there may be a need for exceptions to the default. These can be handled by aliasing them to *:unknown:*. This differs from *:fail:* in that it causes the *redirect* router to decline, whereas *:fail:* forces routing to fail. A lookup which results in an empty redirection list has the same effect.

22.7 Duplicate addresses

Exim removes duplicate addresses from the list to which it is delivering, so as to deliver just one copy to each address. This does not apply to deliveries routed to pipes by different immediate parent addresses, but an indirect aliasing scheme of the type

```
pipe:       |/some/command $local_part
localpart1: pipe
localpart2: pipe
```

does not work with a message that is addressed to both local parts, because when the second is aliased to the intermediate local part "pipe" it gets discarded as being the same as a previously handled address. However, a scheme such as

```
localpart1: |/some/command $local_part
localpart2: |/some/command $local_part
```

does result in two different pipe deliveries, because the immediate parents of the pipes are distinct.

22.8 Repeated redirection expansion

When a message cannot be delivered to all of its recipients immediately, leading to two or more delivery attempts, redirection expansion is carried out afresh each time for those addresses whose children were not all previously delivered. If redirection is being used as a mailing list, this can lead to new members of the list receiving copies of old messages. The **one_time** option can be used to avoid this.

22.9 Errors in redirection lists

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. Otherwise, if an error is detected while generating the list of new addresses, the original address is deferred. See also **syntax_errors_to**.

22.10 Private options for the redirect router

The private options for the *redirect* router are as follows:

allow_defer	Use: *redirect*	Type: *boolean*	Default: *false*

Setting this option allows the use of *:defer:* in non-filter redirection data, or the **defer** command in an Exim filter file.

allow_fail	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is true, the *:fail:* item can be used in a redirection list, and the **fail** command may be used in an Exim filter file.

allow_filter	Use: *redirect*	Type: *boolean*	Default: *false*

Setting this option allows Exim to interpret redirection data that starts with "#Exim filter" or "#Sieve filter" as a set of filtering instructions. There are some features of Exim filter files that some administrators may wish to lock out; see the **forbid_filter_***xxx* options below.

It is also possible to lock out Exim filters or Sieve filters while allowing the other type; see **forbid_exim_filter** and **forbid_sieve_filter**.

The filter is run using the uid and gid set by the generic **user** and **group** options. These take their defaults from the password data if **check_local_user** is set, so in the normal case of users' personal

filter files, the filter is run as the relevant user. When **allow_filter** is set true, Exim insists that either **check_local_user** or **user** is set.

allow_freeze	Use: *redirect*	Type: *boolean*	Default: *false*

Setting this option allows the use of the **freeze** command in an Exim filter. This command is more normally encountered in system filters, and is disabled by default for redirection filters because it isn't something you usually want to let ordinary users do.

check_ancestor	Use: *redirect*	Type: *boolean*	Default: *false*

This option is concerned with handling generated addresses that are the same as some address in the list of redirection ancestors of the current address. Although it is turned off by default in the code, it is set in the default configuration file for handling users' *.forward* files. It is recommended for this use of the *redirect* router.

When **check_ancestor** is set, if a generated address (including the domain) is the same as any ancestor of the current address, it is replaced by a copy of the current address. This helps in the case where local part A is aliased to B, and B has a *.forward* file pointing back to A. For example, within a single domain, the local part "Joe.Bloggs" is aliased to "jb" and *jb/.forward* contains:

```
\Joe.Bloggs, <other item(s)>
```

Without the **check_ancestor** setting, either local part ("jb" or "joe.bloggs") gets processed once by each router and so ends up as it was originally. If "jb" is the real mailbox name, mail to "jb" gets delivered (having been turned into "joe.bloggs" by the *.forward* file and back to "jb" by the alias), but mail to "joe.bloggs" fails. Setting **check_ancestor** on the *redirect* router that handles the *.forward* file prevents it from turning "jb" back into "joe.bloggs" when that was the original address. See also the **repeat_use** option below.

check_group	Use: *redirect*	Type: *boolean*	Default: *see below*

When the **file** option is used, the group owner of the file is checked only when this option is set. The permitted groups are those listed in the **owngroups** option, together with the user's default group if **check_local_user** is set. If the file has the wrong group, routing is deferred. The default setting for this option is true if **check_local_user** is set and the **modemask** option permits the group write bit, or if the **owngroups** option is set. Otherwise it is false, and no group check occurs.

check_owner	Use: *redirect*	Type: *boolean*	Default: *see below*

When the **file** option is used, the owner of the file is checked only when this option is set. If **check_local_user** is set, the local user is permitted; otherwise the owner must be one of those listed in the **owners** option. The default value for this option is true if **check_local_user** or **owners** is set. Otherwise the default is false, and no owner check occurs.

data	Use: *redirect*	Type: *string*†	Default: *unset*

This option is mutually exclusive with **file**. One or other of them must be set, but not both. The contents of **data** are expanded, and then used as the list of forwarding items, or as a set of filtering instructions. If the expansion is forced to fail, or the result is an empty string or a string that has no effect (consists entirely of comments), the router declines.

When filtering instructions are used, the string must begin with "#Exim filter", and all comments in the string, including this initial one, must be terminated with newline characters. For example:

```
data = #Exim filter\n\
       if $h_to: contains Exim then save $home/mail/exim endif
```

If you are reading the data from a database where newlines cannot be included, you can use the *${sg}* expansion item to turn the escape string of your choice into a newline.

directory_transport	Use: *redirect*	Type: *string*†	Default: *unset*

A *redirect* router sets up a direct delivery to a directory when a path name 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. This should normally be an *appendfile* transport.

file	Use: *redirect*	Type: *string*†	Default: *unset*

This option specifies the name of a file that contains the redirection data. It is mutually exclusive with the **data** option. The string is expanded before use; if the expansion is forced to fail, the router declines. Other expansion failures cause delivery to be deferred. The result of a successful expansion must be an absolute path. The entire file is read and used as the redirection data. If the data is an empty string or a string that has no effect (consists entirely of comments), the router declines.

If the attempt to open the file fails with a "does not exist" error, Exim runs a check on the containing directory, unless **ignore_enotdir** is true (see below). If the directory does not appear to exist, delivery is deferred. This can happen when users' *.forward* files are in NFS-mounted directories, and there is a mount problem. If the containing directory does exist, but the file does not, the router declines.

file_transport	Use: *redirect*	Type: *string*†	Default: *unset*

A *redirect* router sets up a direct delivery to a file when a path name 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. This should normally be an *appendfile* transport. When it is running, the file name is in *$address_file*.

filter_prepend_home	Use: *redirect*	Type: *boolean*	Default: *true*

When this option is true, if a *save* command in an Exim filter specifies a relative path, and *$home* is defined, it is automatically prepended to the relative path. If this option is set false, this action does not happen. The relative path is then passed to the transport unmodified.

forbid_blackhole	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is true, the *:blackhole:* item may not appear in a redirection list.

forbid_exim_filter	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is set true, only Sieve filters are permitted when **allow_filter** is true.

forbid_file	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is true, this router may not generate a new address that specifies delivery to a local file or directory, either from a filter or from a conventional forward file. This option is forced to be true if **one_time** is set. It applies to Sieve filters as well as to Exim filters, but if true, it locks out the Sieve's "keep" facility.

forbid_filter_dlfunc	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is true, string expansions in Exim filters are not allowed to make use of the **dlfunc** expansion facility to run dynamically loaded functions.

forbid_filter_existstest	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is true, string expansions in Exim filters are not allowed to make use of the **exists** condition or the **stat** expansion item.

forbid_filter_logwrite	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is true, use of the logging facility in Exim filters is not permitted. Logging 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	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is true, string expansions in Exim filter files are not allowed to make use of **lookup** items.

forbid_filter_perl	Use: *redirect*	Type: *boolean*	Default: *false*

This option has an effect only if Exim is built with embedded Perl support. If it is true, string expansions in Exim filter files are not allowed to make use of the embedded Perl support.

forbid_filter_readfile	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is true, string expansions in Exim filter files are not allowed to make use of **readfile** items.

forbid_filter_readsocket	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is true, string expansions in Exim filter files are not allowed to make use of **readsocket** items.

forbid_filter_reply	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is true, this router may not generate an automatic reply message. Automatic replies can be generated only from Exim or Sieve filter files, not from traditional forward files. This option is forced to be true if **one_time** is set.

forbid_filter_run	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is true, string expansions in Exim filter files are not allowed to make use of **run** items.

forbid_include	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is true, items of the form

```
:include:<path name>
```

are not permitted in non-filter redirection lists.

forbid_pipe	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is true, this router may not generate a new address which specifies delivery to a pipe, either from an Exim filter or from a conventional forward file. This option is forced to be true if **one_time** is set.

forbid_sieve_filter	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is set true, only Exim filters are permitted when **allow_filter** is true.

forbid_smtp_code	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is set true, any SMTP error codes that are present at the start of messages specified for `:defer:` or `:fail:` are quietly ignored, and the default codes (451 and 550, respectively) are always used.

hide_child_in_errmsg	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is true, it prevents Exim from quoting a child address if it generates a bounce or delay message for it. Instead it says "an address generated from <*the top level address*>". Of course, this applies only to bounces generated locally. If a message is forwarded to another host, *its* bounce may well quote the generated address.

ignore_eacces	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is set and an attempt to open a redirection file yields the EACCES error (permission denied), the *redirect* router behaves as if the file did not exist.

ignore_enotdir	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is set and an attempt to open a redirection file yields the ENOTDIR error (something on the path is not a directory), the *redirect* router behaves as if the file did not exist.

Setting **ignore_enotdir** has another effect as well: When a *redirect* router that has the **file** option set discovers that the file does not exist (the ENOENT error), it tries to *stat()* the parent directory, as a check against unmounted NFS directories. If the parent can not be statted, delivery is deferred. However, it seems wrong to do this check when **ignore_enotdir** is set, because that option tells Exim to ignore "something on the path is not a directory" (the ENOTDIR error). This is a confusing area, because it seems that some operating systems give ENOENT where others give ENOTDIR.

include_directory	Use: *redirect*	Type: *string*	Default: *unset*

If this option is set, the path names of any *:include:* items in a redirection list must start with this directory.

modemask	Use: *redirect*	Type: *octal integer*	Default: *022*

This specifies mode bits which must not be set for a file specified by the **file** option. If any of the forbidden bits are set, delivery is deferred.

one_time	Use: *redirect*	Type: *boolean*	Default: *false*

Sometimes the fact that Exim re-evaluates aliases and reprocesses redirection files each time it tries to deliver a message causes a problem when one or more of the generated addresses fails be delivered at the first attempt. The problem is not one of duplicate delivery – Exim is clever enough to handle that – but of what happens when the redirection list changes during the time that the message is on Exim's queue. This is particularly true in the case of mailing lists, where new subscribers might receive copies of messages that were posted before they subscribed.

If **one_time** is set and any addresses generated by the router 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, redirection does not happen again at the next delivery attempt.

Warning 1: Any header line addition or removal that is specified by this router would be lost if delivery did not succeed at the first attempt. For this reason, the **headers_add** and **headers_remove** generic options are not permitted when **one_time** is set.

Warning 2: To ensure that the router generates only addresses (as opposed to pipe or file deliveries or auto-replies) **forbid_file**, **forbid_pipe**, and **forbid_filter_reply** are forced to be true when **one_time** is set.

Warning 3: The **unseen** generic router option may not be set with **one_time**.

The original top-level address is remembered with each of the generated addresses, and is output in any log messages. However, any intermediate parent addresses are not recorded. This makes a difference to the log only if **all_parents** log selector is set. It is expected that **one_time** will typically be used for mailing lists, where there is normally just one level of expansion.

owners	Use: *redirect*	Type: *string list*	Default: *unset*

This specifies a list of permitted owners for the file specified by **file**. This list is in addition to the local user when **check_local_user** is set. See **check_owner** above.

owngroups	Use: *redirect*	Type: *string list*	Default: *unset*

This specifies a list of permitted groups for the file specified by **file**. The list is in addition to the local user's primary group when **check_local_user** is set. See **check_group** above.

pipe_transport	Use: *redirect*	Type: *string*†	Default: *unset*

A *redirect* router sets up a direct 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. This should normally be a *pipe* transport. When the transport is run, the pipe command is in *$address_pipe*.

qualify_domain	Use: *redirect*	Type: *string*†	Default: *unset*

If this option is set, and an unqualified address (one without a domain) is generated, and that address would normally be qualified by the global setting in **qualify_recipient**, it is instead qualified with the domain specified by expanding this string. If the expansion fails, the router declines. If you want to revert to the default, you can have the expansion generate *$qualify_recipient*.

This option applies to all unqualified addresses generated by Exim filters, but for traditional *.forward* files, it applies only to addresses that are not preceded by a backslash. Sieve filters cannot generate unqualified addresses.

qualify_preserve_domain	Use: *redirect*	Type: *boolean*	Default: *false*

If this option is set, the router's local **qualify_domain** option must not be set (a configuration error occurs if it is). If an unqualified address (one without a domain) is generated, it is qualified with the domain of the parent address (the immediately preceding ancestor) instead of the global **qualify_ recipient** value. In the case of a traditional *.forward* file, this applies whether or not the address is preceded by a backslash.

repeat_use	Use: *redirect*	Type: *boolean*	Default: *true*

If this option is set false, the router is skipped for a child address that has any ancestor that was routed by this router. This test happens before any of the other preconditions are tested. Exim's default anti-looping rules skip only when the ancestor is the same as the current address. See also **check_ ancestor** above and the generic **redirect_router** option.

reply_transport	Use: *redirect*	Type: *string*†	Default: *unset*

A *redirect* router sets up an automatic reply 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. This should normally be an *autoreply* transport. Other transports are unlikely to do anything sensible or useful.

rewrite	Use: *redirect*	Type: *boolean*	Default: *true*

If this option is set false, addresses generated by the router are not subject to address rewriting. Otherwise, they are treated like new addresses and are rewritten according to the global rewriting rules.

sieve_subaddress	Use: *redirect*	Type: *string*†	Default: *unset*

The value of this option is passed to a Sieve filter to specify the :subaddress part of an address.

sieve_useraddress	Use: *redirect*	Type: *string*†	Default: *unset*

The value of this option is passed to a Sieve filter to specify the :user part of an address. However, if it is unset, the entire original local part (including any prefix or suffix) is used for :user.

sieve_vacation_directory	Use: *redirect*	Type: *string*†	Default: *unset*

To enable the "vacation" extension for Sieve filters, you must set **sieve_vacation_directory** to the directory where vacation databases are held (do not put anything else in that directory), and ensure that the **reply_transport** option refers to an *autoreply* transport. Each user needs their own directory; Exim will create it if necessary.

skip_syntax_errors	Use: *redirect*	Type: *boolean*	Default: *false*

If **skip_syntax_errors** is set, syntactically malformed addresses in non-filter redirection data are skipped, and each failing address is logged. If **syntax_errors_to** is set, a message is sent to the address it defines, giving details of the failures. If **syntax_errors_text** is set, its contents are expanded and placed at the head of the error message generated by **syntax_errors_to**. Usually it is appropriate to set **syntax_errors_to** to be the same address as the generic **errors_to** option. The **skip_syntax_ errors** option is often used when handling mailing lists.

If all the addresses in a redirection list are skipped because of syntax errors, the router declines to handle the original address, and it is passed to the following routers.

If **skip_syntax_errors** is set when an Exim filter is interpreted, any syntax error in the filter causes filtering to be abandoned without any action being taken. The incident is logged, and the router declines to handle the address, so it is passed to the following routers.

Syntax errors in a Sieve filter file cause the "keep" action to occur. This action is specified by RFC 3028. The values of **skip_syntax_errors**, **syntax_errors_to**, and **syntax_errors_text** are not used.

skip_syntax_errors can be used to specify that errors in users' forward lists or filter files should not prevent delivery. The **syntax_errors_to** option, used with an address that does not get redirected, can be used to notify users of these errors, by means of a router like this:

```
userforward:
  driver = redirect
  allow_filter
  check_local_user
  file = $home/.forward
  file_transport = address_file
  pipe_transport = address_pipe
  reply_transport = address_reply
  no_verify
  skip_syntax_errors
  syntax_errors_to = real-$local_part@$domain
  syntax_errors_text = \
   This is an automatically generated message. An error has\n\
   been found in your .forward file. Details of the error are\n\
   reported below. While this error persists, you will receive\n\
   a copy of this message for every message that is addressed\n\
   to you. If your .forward file is a filter file, or if it is\n\
   a non-filter file containing no valid forwarding addresses,\n\
   a copy of each incoming message will be put in your normal\n\
   mailbox. If a non-filter file contains at least one valid\n\
   forwarding address, forwarding to the valid addresses will\n\
   happen, and those will be the only deliveries that occur.
```

You also need a router to ensure that local addresses that are prefixed by `real-` are recognized, but not forwarded or filtered. For example, you could put this immediately before the *userforward* router:

```
real_localuser:
  driver = accept
  check_local_user
  local_part_prefix = real-
  transport = local_delivery
```

For security, it would probably be a good idea to restrict the use of this router to locally-generated messages, using a condition such as this:

```
condition = ${if match {$sender_host_address}\
                       {\N^(|127\.0\.0\.1)$\N}}
```

syntax_errors_text	Use: *redirect*	Type: *string*†	Default: *unset*

See **skip_syntax_errors** above.

syntax_errors_to	Use: *redirect*	Type: *string*	Default: *unset*

See **skip_syntax_errors** above.

23. Environment for running local transports

Local transports handle deliveries to files and pipes. (The *autoreply* transport can be thought of as similar to a pipe.) Exim always runs transports in subprocesses, under specified uids and gids. Typical deliveries to local mailboxes run under the uid and gid of the local user.

Exim also sets a specific current directory while running the transport; for some transports a home directory setting is also relevant. The *pipe* transport is the only one that sets up environment variables; see section 29.4 for details.

The values used for the uid, gid, and the directories may come from several different places. In many cases, the router that handles the address associates settings with that address as a result of its **check_local_user**, **group**, or **user** options. However, values may also be given in the transport's own configuration, and these override anything that comes from the router.

23.1 Concurrent deliveries

If two different messages for the same local recipient arrive more or less simultaneously, the two delivery processes are likely to run concurrently. When the *appendfile* transport is used to write to a file, Exim applies locking rules to stop concurrent processes from writing to the same file at the same time.

However, when you use a *pipe* transport, it is up to you to arrange any locking that is needed. Here is a silly example:

```
my_transport:
  driver = pipe
  command = /bin/sh -c 'cat >>/some/file'
```

This is supposed to write the message at the end of the file. However, if two messages arrive at the same time, the file will be scrambled. You can use the **exim_lock** utility program (see section 50.15) to lock a file using the same algorithm that Exim itself uses.

23.2 Uids and gids

All transports have the options **group** and **user**. If **group** is set, it overrides any group that the router set in the address, even if **user** is not set for the transport. This makes it possible, for example, to run local mail delivery under the uid of the recipient (set by the router), but in a special group (set by the transport). For example:

```
# Routers ...
# User/group are set by check_local_user in this router
local_users:
  driver = accept
  check_local_user
  transport = group_delivery

# Transports ...
# This transport overrides the group
group_delivery:
  driver = appendfile
  file = /var/spool/mail/$local_part
  group = mail
```

If **user** is set for a transport, its value overrides what is set in the address by the router. If **user** is non-numeric and **group** is not set, the gid associated with the user is used. If **user** is numeric, **group** must be set.

When the uid is taken from the transport's configuration, the *initgroups()* function is called for the groups associated with that uid if the **initgroups** option is set for the transport. When the uid is not

specified by the transport, but is associated with the address by a router, the option for calling *initgroups()* is taken from the router configuration.

The *pipe* transport contains the special option **pipe_as_creator**. If this is set and **user** is not set, the uid of the process that called Exim to receive the message is used, and if **group** is not set, the corresponding original gid is also used.

This is the detailed preference order for obtaining a gid; the first of the following that is set is used:

- A **group** setting of the transport;
- A **group** setting of the router;
- A gid associated with a user setting of the router, either as a result of **check_local_user** or an explicit non-numeric **user** setting;
- The group associated with a non-numeric **user** setting of the transport;
- In a *pipe* transport, the creator's gid if **deliver_as_creator** is set and the uid is the creator's uid;
- The Exim gid if the Exim uid is being used as a default.

If, for example, the user is specified numerically on the router and there are no group settings, no gid is available. In this situation, an error occurs. This is different for the uid, for which there always is an ultimate default. The first of the following that is set is used:

- A **user** setting of the transport;
- In a *pipe* transport, the creator's uid if **deliver_as_creator** is set;
- A **user** setting of the router;
- A **check_local_user** setting of the router;
- The Exim uid.

Of course, an error will still occur if the uid that is chosen is on the **never_users** list.

23.3 Current and home directories

Routers may set current and home directories for local transports by means of the **transport_current_directory** and **transport_home_directory** options. However, if the transport's **current_directory** or **home_directory** options are set, they override the router's values. In detail, the home directory for a local transport is taken from the first of these values that is set:

- The **home_directory** option on the transport;
- The **transport_home_directory** option on the router;
- The password data if **check_local_user** is set on the router;
- The **router_home_directory** option on the router.

The current directory is taken from the first of these values that is set:

- The **current_directory** option on the transport;
- The **transport_current_directory** option on the router.

If neither the router nor the transport sets a current directory, Exim uses the value of the home directory, if it is set. Otherwise it sets the current directory to `/` before running a local transport.

23.4 Expansion variables derived from the address

Normally a local delivery is handling a single address, and in that case the variables such as *$domain* and *$local_part* are set during local deliveries. However, in some circumstances more than one address may be handled at once (for example, while writing 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.

24. Generic options for transports

The following generic options apply to all transports:

body_only	Use: *transports*	Type: *boolean*	Default: *false*

If this option is set, the message's headers are not transported. It is mutually exclusive with **headers_only**. If it is used with the *appendfile* or *pipe* transports, the settings of **message_prefix** and **message_suffix** should be checked, because this option does not automatically suppress them.

current_directory	Use: *transports*	Type: *string*†	Default: *unset*

This specifies the current directory that is to be set while running the transport, overriding any value that may have been set by the router. If the expansion fails for any reason, including forced failure, an error is logged, and delivery is deferred.

disable_logging	Use: *transports*	Type: *boolean*	Default: *false*

If this option is set true, nothing is logged for any deliveries by the transport or for any transport errors. You should not set this option unless you really, really know what you are doing.

debug_print	Use: *transports*	Type: *string*†	Default: *unset*

If this option is set and debugging is enabled (see the **-d** command line option), the string is expanded and included in the debugging output when the transport is run. If expansion of the string fails, the error message is written to the debugging output, and Exim carries on processing. This facility is provided to help with checking out the values of variables and so on when debugging driver configurations. For example, if a **headers_add** option is not working properly, **debug_print** could be used to output the variables it references. A newline is added to the text if it does not end with one.

delivery_date_add	Use: *transports*	Type: *boolean*	Default: *false*

If this option is true, a *Delivery-date:* header is added to the message. This gives the actual time the delivery was made. As this is not a standard header, Exim has a configuration option (**delivery_date_remove**) which requests its removal from incoming messages, so that delivered messages can safely be resent to other recipients.

driver	Use: *transports*	Type: *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	Use: *transports*	Type: *boolean*	Default: *false*

If this option is true, an *Envelope-to:* header is added to the message. This gives the original address(es) in the incoming envelope that caused this delivery to happen. More than one address may be present if the transport is configured to handle several addresses at once, or if more than one original address was redirected to the same final address. As this is not a standard header, Exim has a configuration option (**envelope_to_remove**) which requests its removal from incoming messages, so that delivered messages can safely be resent to other recipients.

group	Use: *transports*	Type: *string*†	Default: *Exim group*

This option specifies a gid for running the transport process, overriding any value that the router supplies, and also overriding any value associated with **user** (see below).

headers_add	Use: *transports*	Type: *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, as described in section 44.17. Additional header lines can also be specified by routers. 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	Use: *transports*	Type: *boolean*	Default: *false*

If this option is set, the message's body is not transported. It is mutually exclusive with **body_only**. If it is used with the *appendfile* or *pipe* transports, the settings of **message_prefix** and **message_suffix** should be checked, since this option does not automatically suppress them.

headers_remove	Use: *transports*	Type: *string*†	Default: *unset*

This option specifies a string that is expanded into a list of header names; these headers are omitted from the message as it is transported, as described in section 44.17. Header removal can also be specified by routers. 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_rewrite	Use: *transports*	Type: *string*	Default: *unset*

This option allows addresses in header lines to be rewritten at transport time, that is, as the message is being copied to its destination. The contents of the option are 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. These are described in chapter 31. 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. The rules are applied to the header lines just before they are written out at transport time, so they affect only those copies of the message that pass through the transport. However, only the message's original header lines, and any that were added by a system filter, are rewritten. If a router or transport adds header lines, they are not affected by this option. These rewriting rules are *not* applied to the envelope. You can change the return path using **return_path**, but you cannot change envelope recipients at this time.

home_directory	Use: *transports*	Type: *string*†	Default: *unset*

This option specifies a home directory setting for a local transport, overriding any value that may be set by the router. The home directory is placed in *$home* while expanding the transport's private options. It is also used as the current directory if no current directory is set by the **current_directory** option on the transport or the **transport_current_directory** option on the router. If the expansion fails for any reason, including forced failure, an error is logged, and delivery is deferred.

initgroups	Use: *transports*	Type: *boolean*	Default: *false*

If this option is true and the uid for the delivery process is provided by the transport, the *initgroups()* function is called when running the transport to ensure that any additional groups associated with the uid are set up.

message_size_limit	Use: *transports*	Type: *string*†	Default: *0*

This option controls the size of messages passed through the transport. It is expanded before use; the result of the expansion must be a sequence of decimal digits, optionally followed by K or M. If the expansion fails for any reason, including forced failure, or if the result is not of the required form, delivery is deferred. If the value is greater than zero and the size of a message exceeds this limit, the address is failed. If there is any chance that the resulting bounce message could be routed to the same transport, you should ensure that **return_size_limit** is less than the transport's **message_size_limit**, as otherwise the bounce message will fail to get delivered.

rcpt_include_affixes	Use: *transports*	Type: *boolean*	Default: *false*

When this option is false (the default), and an address that has had any affixes (prefixes or suffixes) removed from the local part is delivered by any form of SMTP or LMTP, the affixes are not included. For example, if a router that contains

```
local_part_prefix = *-
```

routes the address *abc-xyz@some.domain* to an SMTP transport, the envelope is delivered with

```
RCPT TO:<xyz@some.domain>
```

This is also the case when an ACL-time callout is being used to verify a recipient address. However, if **rcpt_include_affixes** is set true, the whole local part is included in the RCPT command. This option applies to BSMTP deliveries by the *appendfile* and *pipe* transports as well as to the *lmtp* and *smtp* transports.

retry_use_local_part	Use: *transports*	Type: *boolean*	Default: *see below*

When a delivery suffers a temporary failure, a retry record is created in Exim's hints database. For remote deliveries, the key for the retry record is based on the name and/or IP address of the failing remote host. For local deliveries, the key is normally the entire address, including both the local part and the domain. This is suitable for most common cases of local delivery temporary failure – for example, exceeding a mailbox quota should delay only deliveries to that mailbox, not to the whole domain.

However, in some special cases you may want to treat a temporary local delivery as a failure associated with the domain, and not with a particular local part. (For example, if you are storing all mail for some domain in files.) You can do this by setting **retry_use_local_part** false.

For all the local transports, its default value is true. For remote transports, the default value is false for tidiness, but changing the value has no effect on a remote transport in the current implementation.

return_path	Use: *transports*	Type: *string*†	Default: *unset*

If this option is set, the string is expanded at transport time and replaces the existing return path (envelope sender) value in the copy of the message that is being delivered. An empty return path is permitted. This feature is designed for remote deliveries, where the value of this option is used in the SMTP MAIL command. If you set **return_path** for a local transport, the only effect is to change the address that is placed in the *Return-path:* header line, if one is added to the message (see the next option).

Note: A changed return path is not logged unless you add **return_path_on_delivery** to the log selector.

The expansion can refer to the existing value via *$return_path*. This is either the message's envelope sender, or an address set by the **errors_to** option on a router. If the expansion is forced to fail, no replacement occurs; if it fails for another reason, delivery is deferred. This option can be used to support VERP (Variable Envelope Return Paths) – see section 47.6.

Note: If a delivery error is detected locally, including the case when a remote server rejects a message at SMTP time, the bounce message is not sent to the value of this option. It is sent to the previously set errors address. This defaults to the incoming sender address, but can be changed by setting **errors_to** in a router.

return_path_add	Use: *transports*	Type: *boolean*	Default: *false*

If this option is true, a *Return-path:* header is added to the message. Although the return path is normally available in the prefix line of BSD mailboxes, this is commonly not displayed by MUAs, and so the user does not have easy access to it.

RFC 2821 states that the *Return-path:* header is added to a message "when the delivery SMTP server makes the final delivery". This implies that this header should not be present in incoming messages. Exim has a configuration option, **return_path_remove**, which requests removal of this header from incoming messages, so that delivered messages can safely be resent to other recipients.

shadow_condition	Use: *transports*	Type: *string*†	Default: *unset*

See **shadow_transport** below.

shadow_transport	Use: *transports*	Type: *string*	Default: *unset*

A local transport may set the **shadow_transport** option to the name of another local transport. Shadow remote transports are not supported.

Whenever a delivery to the main transport succeeds, and either **shadow_condition** is unset, or its expansion does not result in the empty string or one of the strings "0" or "no" or "false", the message is also passed to the shadow transport, with the same delivery address or addresses. If expansion fails, no action is taken except that non-forced expansion failures cause a log line to be written.

The result of the shadow transport is discarded and does not affect the subsequent processing of the message. 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, of the form

```
ST=<shadow transport name>
```

If the shadow transport did not succeed, the error message is put in parentheses afterwards. 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 that some sites insist on.

transport_filter	Use: *transports*	Type: *string*†	Default: *unset*

This option sets up a filtering (in the Unix shell sense) process for messages at transport time. It should not be confused with mail filtering as set up by individual users or via a system filter.

When the message is about to be written out, the command specified by **transport_filter** is started up in a separate, parallel process, and the entire message, including the header lines, is passed to it on its standard input (this in fact is done from a third process, to avoid deadlock). The command must be specified as an absolute path.

The lines of the message that are written to the transport filter are terminated by newline ("\n"). The message is passed to the filter before any SMTP-specific processing, such as turning "\n" into "\r\n" and escaping lines beginning with a dot, and also before any processing implied by the settings of **check_string** and **escape_string** in the *appendfile* or *pipe* transports.

The standard error for the filter process is set to the same destination as its standard output; this is read and written to the message's ultimate destination. The process that writes the message to the filter, the filter itself, and the original process that reads the result and delivers it are all run in parallel, like a shell pipeline.

The filter can perform any transformations it likes, but of course should take care not to break RFC 2822 syntax. A demonstration Perl script is provided in *util/transport-filter.pl*; this makes a few arbitrary modifications just to show the possibilities. Exim does not check the result, except to test for a final newline when SMTP is in use. All messages transmitted over SMTP must end with a newline, so Exim supplies one if it is missing.

A transport filter can be used to provide content-scanning on a per-user basis at delivery time if the only required effect of the scan is to modify the message. For example, a content scan could insert a new header line containing a spam score. This could be interpreted by a filter in the user's MUA. It is not possible to discard a message at this stage.

A problem might arise if the filter increases the size of a message that is being sent down an SMTP connection. 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. This can be worked round 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 the **transport_filter** option is the command string for starting the filter, which is run directly from Exim, 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 then expands each argument separately (see section 29.3). Any kind of expansion failure causes delivery to be deferred. The special argument *$pipe_addresses* is replaced by a number of arguments, one for each address that applies to this delivery. (This isn't an ideal name for this feature here, but as it was already implemented for the *pipe* transport, it seemed sensible not to change it.)

The expansion variables *$host* and *$host_address* are available when the transport is a remote one. They contain the name and IP address of the host to which the message is being sent. For example:

```
transport_filter = /some/directory/transport-filter.pl \
  $host $host_address $sender_address $pipe_addresses
```

Two problems arise if you want to use more complicated expansion items to generate transport filter commands, both of which due to the fact that the command is split up *before* expansion.

- If an expansion item contains white space, you must quote it, so that it is all part of the same command item. If the entire option setting is one such expansion item, you have to take care what kind of quoting you use. For example:

  ```
  transport_filter = '/bin/cmd${if eq{$host}{a.b.c}{1}{2}}'
  ```

  This runs the command */bin/cmd1* if the host name is *a.b.c*, and */bin/cmd2* otherwise. If double quotes had been used, they would have been stripped by Exim when it read the option's value. When the value is used, if the single quotes were missing, the line would be split into two items, `/bin/cmd${if` and `eq{$host}{a.b.c}{1}{2}`, and an error would occur when Exim tried to expand the first one.

- Except for the special case of *$pipe_addresses* that is mentioned above, an expansion cannot generate multiple arguments, or a command name followed by arguments. Consider this example:

  ```
  transport_filter = ${lookup{$host}lsearch{/a/file}\
                       {$value}{/bin/cat}}
  ```

  The result of the lookup is interpreted as the name of the command, even if it contains white space. The simplest way round this is to use a shell:

```
transport_filter = /bin/sh -c ${lookup{$host}lsearch{/a/file}\
                                {$value}{/bin/cat}}
```

The filter process is run under the same uid and gid as the normal delivery. For remote deliveries this is the Exim uid/gid by default. The command should normally yield a zero return code. Transport filters are not supposed to fail. A non-zero code is taken to mean that the transport filter encountered some serious problem. Delivery of the message is deferred; the message remains on the queue and is tried again later. It is not possible to cause a message to be bounced from a transport filter.

If a transport filter is set on an autoreply transport, the original message is passed through the filter as it is being copied into the newly generated message, which happens if the **return_message** option is set.

transport_filter_timeout	Use: *transports*	Type: *time*	Default: *5m*

When Exim is reading the output of a transport filter, it a applies a timeout that can be set by this option. Exceeding the timeout is normally treated as a temporary delivery failure. However, if a transport filter is used with a *pipe* transport, a timeout in the transport filter is treated in the same way as a timeout in the pipe command itself. By default, a timeout is a hard error, but if the *pipe* transport's **timeout_defer** option is set true, it becomes a temporary error.

user	Use: *transports*	Type: *string*†	Default: *Exim user*

This option specifies the user under whose uid the delivery process is to be run, overriding any uid that may have been set by the router. If the user is given as a name, the uid is looked up from the password data, and the associated group is taken as the value of the gid to be used if the **group** option is not set.

For deliveries that use local transports, a user and group are normally specified explicitly or implicitly (for example, as a result of **check_local_user**) by the router or transport.

For remote transports, you should leave this option unset unless you really are sure you know what you are doing. When a remote transport is running, it needs to be able to access Exim's hints databases, because each host may have its own retry data.

25. Address batching in local transports

The only remote transport (*smtp*) is normally configured to handle more than one address at a time, so that when several addresses are routed to the same remote host, just one copy of the message is sent. Local transports, however, normally handle one address at a time. That is, a separate instance of the transport is run for each address that is routed to the transport. A separate copy of the message is delivered each time.

In special cases, it may be desirable to handle several addresses at once in a local transport, for example:

- In an *appendfile* transport, when storing messages in files for later delivery by some other means, a single copy of the message with multiple recipients saves space.
- In an *lmtp* transport, when delivering over "local SMTP" to some process, a single copy saves time, and is the normal way LMTP is expected to work.
- In a *pipe* transport, when passing the message to a scanner program or to some other delivery mechanism such as UUCP, multiple recipients may be acceptable.

These three local transports all have the same options for controlling multiple ("batched") deliveries, namely **batch_max** and **batch_id**. To save repeating the information for each transport, these options are described here.

The **batch_max** option specifies the maximum number of addresses that can be delivered together in a single run of the transport. Its default value is one (no batching). When more than one address is routed to a transport that has a **batch_max** value greater than one, the addresses are delivered in a batch (that is, in a single run of the transport with multiple recipients), subject to certain conditions:

- If any of the transport's options contain a reference to *$local_part*, no batching is possible.
- If any of the transport's options contain a reference to *$domain*, only addresses with the same domain are batched.
- If **batch_id** is set, it is expanded for each address, and only those addresses with the same expanded value are batched. This allows you to specify customized batching conditions. Failure of the expansion for any reason, including forced failure, disables batching, but it does not stop the delivery from taking place.
- Batched addresses must also have the same errors address (where to send delivery errors), the same header additions and removals, the same user and group for the transport, and if a host list is present, the first host must be the same.

In the case of the *appendfile* and *pipe* transports, batching applies both when the file or pipe command is specified in the transport, and when it is specified by a *redirect* router, but all the batched addresses must of course be routed to the same file or pipe command. These two transports have an option called **use_bsmtp**, which causes them to deliver the message in "batched SMTP" format, with the envelope represented as SMTP commands. The **check_string** and **escape_string** options are forced to the values

```
check_string = "."
escape_string = ".."
```

when batched SMTP is in use. A full description of the batch SMTP mechanism is given in section 45.10. The *lmtp* transport does not have a **use_bsmtp** option, because it always delivers using the SMTP protocol.

If the generic **envelope_to_add** option is set for a batching transport, the *Envelope-to:* header that is added to the message contains all the addresses that are being processed together. If you are using a batching *appendfile* transport without **use_bsmtp**, the only way to preserve the recipient addresses is to set the **envelope_to_add** option.

If you are using a *pipe* transport without BSMTP, and setting the transport's **command** option, you can include *$pipe_addresses* as part of the command. This is not a true variable; it is a bit of magic

that causes each of the recipient addresses to be inserted into the command as a separate argument. This provides a way of accessing all the addresses that are being delivered in the batch. **Note:** This is not possible for pipe commands that are specified by a *redirect* router.

26. The appendfile transport

The *appendfile* transport delivers a message by appending it to an existing file, or by creating an entirely new file in a specified directory. Single files to which messages are appended can be in the traditional Unix mailbox format, or optionally in the MBX format supported by the Pine MUA and University of Washington IMAP daemon, *inter alia*. When each message is being delivered as a separate file, "maildir" format can optionally be used to give added protection against failures that happen part-way through the delivery. A third form of separate-file delivery known as "mailstore" is also supported. For all file formats, Exim attempts to create as many levels of directory as necessary, provided that **create_directory** is set.

The code for the optional formats is not included in the Exim binary by default. It is necessary to set SUPPORT_MBX, SUPPORT_MAILDIR and/or SUPPORT_MAILSTORE in *Local/Makefile* to have the appropriate code included.

Exim recognizes system quota errors, and generates an appropriate message. Exim also supports its own quota control within the transport, for use when the system facility is unavailable or cannot be used for some reason.

If there is an error while appending to a file (for example, quota exceeded or partition filled), Exim attempts to reset the file's length and last modification time back to what they were before. If there is an error while creating an entirely new file, the new file is removed.

Before appending to a file, a number of security checks are made, and the file is locked. A detailed description is given below, after the list of private options.

The *appendfile* transport is most commonly used for local deliveries to users' mailboxes. However, it can also be used as a pseudo-remote transport for putting messages into files for remote delivery by some means other than Exim. "Batch SMTP" format is often used in this case (see the **use_bsmtp** option).

26.1 The file and directory options

The **file** option specifies a single file, to which the message is appended; the **directory** option specifies a directory, in which a new file containing the message is created. Only one of these two options can be set, and for normal deliveries to mailboxes, one of them *must* be set.

However, *appendfile* is also used for delivering messages to files or directories whose names (or parts of names) are obtained from alias, forwarding, or filtering operations (for example, a **save** command in a user's Exim filter). When such a transport is running, *$local_part* contains the local part that was aliased or forwarded, and *$address_file* contains the name (or partial name) of the file or directory generated by the redirection operation. There are two cases:

- If neither **file** nor **directory** is set, the redirection operation must specify an absolute path (one that begins with `/`). This is the most common case when users with local accounts use filtering to sort mail into different folders. See for example, the *address_file* transport in the default configuration. If the path ends with a slash, it is assumed to be the name of a directory. A delivery to a directory can also be forced by setting **maildir_format** or **mailstore_format**.
- If **file** or **directory** is set for a delivery from a redirection, it is used to determine the file or directory name for the delivery. Normally, the contents of *$address_file* are used in some way in the string expansion.

As an example of the second case, consider an environment where users do not have home directories. They may be permitted to use Exim filter commands of the form:

```
save folder23
```

or Sieve filter commands of the form:

```
require "fileinto";
fileinto "folder23";
```

In this situation, the expansion of **file** or **directory** in the transport must transform the relative path into an appropriate absolute file name. In the case of Sieve filters, the name *inbox* must be handled. It is the name that is used as a result of a "keep" action in the filter. This example shows one way of handling this requirement:

```
file = ${if eq{$address_file}{inbox} \
            {/var/mail/$local_part} \
            {${if eq{${substr_0_1:$address_file}}{/} \
                  {$address_file} \
                  {$home/mail/$address_file} \
            }} \
       }
```

With this setting of **file**, *inbox* refers to the standard mailbox location, absolute paths are used without change, and other folders are in the *mail* directory within the home directory.

Note 1: While processing an Exim filter, a relative path such as *folder23* is turned into an absolute path if a home directory is known to the router. In particular, this is the case if **check_local_user** is set. If you want to prevent this happening at routing time, you can set **router_home_directory** empty. This forces the router to pass the relative path to the transport.

Note 2: An absolute path in *$address_file* is not treated specially; the **file** or **directory** option is still used if it is set.

26.2 Private options for appendfile

allow_fifo	Use: *appendfile*	Type: *boolean*	Default: *false*

Setting this option permits delivery to named pipes (FIFOs) as well as to regular files. If no process is reading the named pipe at delivery time, the delivery is deferred.

allow_symlink	Use: *appendfile*	Type: *boolean*	Default: *false*

By default, *appendfile* will not deliver if the path name for the file is that of a symbolic link. Setting this option 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 set this. Details of exactly what this option affects are included in the discussion which follows this list of options.

batch_id	Use: *appendfile*	Type: *string*†	Default: *unset*

See the description of local delivery batching in chapter 25. However, batching is automatically disabled for *appendfile* deliveries that happen as a result of forwarding or aliasing or other redirection directly to a file.

batch_max	Use: *appendfile*	Type: *integer*	Default: *1*

See the description of local delivery batching in chapter 25.

check_group	Use: *appendfile*	Type: *boolean*	Default: *false*

When this option is set, the group owner of the file defined by the **file** option is checked to see that it is the same as the group under which the delivery process is running. The default setting is false because the default file mode is 0600, which means that the group is irrelevant.

check_owner	Use: *appendfile*	Type: *boolean*	Default: *true*

When this option is set, the owner of the file defined by the **file** option is checked to ensure that it is the same as the user under which the delivery process is running.

check_string	Use: *appendfile*	Type: *string*	Default: *see below*

As *appendfile* 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, and the case of any letters it contains is significant.

If **use_bsmtp** is set the values of **check_string** and **escape_string** are forced to "." and ".." respectively, and any settings in the configuration are ignored. Otherwise, they default to "From " and ">From " when the **file** option is set, and unset when any of the **directory**, **maildir**, or **mailstore** options are set.

The default settings, along with **message_prefix** and **message_suffix**, are suitable for traditional "BSD" mailboxes, where a line beginning with "From " indicates the start of a new message. All four options need changing if another format is used. For example, to deliver to mailboxes in MMDF format:

```
check_string = "\1\1\1\1\n"
escape_string = "\1\1\1\1 \n"
message_prefix = "\1\1\1\1\n"
message_suffix = "\1\1\1\1\n"
```

create_directory	Use: *appendfile*	Type: *boolean*	Default: *true*

When this option is true, Exim attempts to create any missing superior directories for the file that it is about to write. A created directory's mode is given by the **directory_mode** option.

The group ownership of a newly created directory is highly dependent on the operating system (and possibly the file system) that is being used. For example, in Solaris, if the parent directory has the setgid bit set, its group is propagated to the child; if not, the currently set group is used. However, in FreeBSD, the parent's group is always used.

create_file	Use: *appendfile*	Type: *string*	Default: *anywhere*

This option constrains the location of files and directories that are created by this transport. It applies to files defined by the **file** option and directories defined by the **directory** option. In the case of maildir delivery, it applies to the top level directory, not the maildir directories beneath.

The option must be set to one of the words "anywhere", "inhome", or "belowhome". In the second and third cases, a home directory must have been set for the transport. This option is not useful when an explicit file name is given for normal mailbox deliveries. It is intended for the case when file names are generated from users' *.forward* files. These are usually handled by an *appendfile* transport called **address_file**. See also **file_must_exist**.

directory	Use: *appendfile*	Type: *string*†	Default: *unset*

This option is mutually exclusive with the **file** option, but one of **file** or **directory** must be set, unless the delivery is the direct result of a redirection (see section 26.1).

When **directory** 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. A number of different formats are provided (see **maildir_format** and **mailstore_format**), and see section 26.4 for further details of this form of delivery.

directory_file	Use: *appendfile*	Type: *string*†	Default: *see below*

When **directory** is set, but neither **maildir_format** nor **mailstore_format** is set, *appendfile* delivers each message into a file whose name is obtained by expanding this string. The default value is:

```
q${base62:$tod_epoch}-$inode
```

This generates a unique name from the current time, in base 62 form, and the inode of the file. The variable *$inode* is available only when expanding this option.

directory_mode	Use: *appendfile*	Type: *octal integer*	Default: *0700*

If *appendfile* creates any directories as a result of the **create_directory** option, their mode is specified by this option.

escape_string	Use: *appendfile*	Type: *string*	Default: *see description*

See **check_string** above.

file	Use: *appendfile*	Type: *string*†	Default: *unset*

This option is mutually exclusive with the **directory** option, but one of **file** or **directory** must be set, unless the delivery is the direct result of a redirection (see section 26.1). The **file** option specifies a single file, to which the message is appended. One or more of **use_fcntl_lock**, **use_flock_lock**, or **use_lockfile** must be set with **file**.

If you are using more than one host to deliver over NFS into the same mailboxes, you should always use lock files.

The string value is expanded for each delivery, and must yield an absolute path. The most common settings of this option are variations on one of these examples:

```
file = /var/spool/mail/$local_part
file = /home/$local_part/inbox
file = $home/inbox
```

In the first example, all deliveries are done into the same directory. If Exim is configured to use lock files (see **use_lockfile** below) it must be able to create a file in the directory, so the "sticky" bit must be turned on for deliveries to be possible, or alternatively the **group** option can be used to run the delivery under a group id which has write access to the directory.

file_format	Use: *appendfile*	Type: *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. The value of the option consists of an even number of colon-separated strings. The first of each pair is the test string, and the second is the name of a transport. If the transport associated with a matched string is not the current transport, control is passed over to the other transport. For example, suppose the standard *local_delivery* transport has this added to it:

```
file_format = "From       : local_delivery :\
               \1\1\1\1\n : local_mmdf_delivery"
```

Mailboxes that begin with "From" are still handled by this transport, but if a mailbox begins with four binary ones followed by a newline, control is passed to a transport called **local_mmdf_delivery**, which presumably is configured to do the delivery in MMDF format. If a mailbox does not exist or is empty, it is assumed to match the current transport. If the start of a mailbox doesn't match any string, or if the transport named for a given string is not defined, delivery is deferred.

file_must_exist	Use: *appendfile*	Type: *boolean*	Default: *false*

If this option is true, the file specified by the **file** option must exist. A temporary error occurs if it does not, causing delivery to be deferred. If this option is false, the file is created if it does not exist.

lock_fcntl_timeout	Use: *appendfile*	Type: *time*	Default: *0s*

By default, the *appendfile* transport uses non-blocking calls to *fcntl()* when locking an open mailbox file. If the call fails, the delivery process sleeps for **lock_interval** and tries again, up to **lock_retries** times. Non-blocking 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 deliveries over NFS in the case when processes might be accessing an NFS mailbox without using a lock file. This should not be done, but misunderstandings and hence misconfigurations are not unknown.

On a busy system, however, the performance of a non-blocking lock approach is not as good as using a blocking lock with a timeout. In this case, the waiting is done inside the system call, and Exim's delivery process acquires the lock and can proceed as soon as the previous lock holder releases it.

If **lock_fcntl_timeout** is set to a non-zero time, blocking locks, with that timeout, are used. There may still be some retrying: the maximum number of retries is

```
(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, unless **lock_fcntl_timeout** is set very large.

You should consider setting this option if you are getting a lot of delayed local deliveries because of errors of the form

```
failed to lock mailbox /some/file (fcntl)
```

lock_flock_timeout	Use: *appendfile*	Type: *time*	Default: *0s*

This timeout applies to file locking when using *flock()* (see **use_flock**); the timeout operates in a similar manner to **lock_fcntl_timeout**.

lock_interval	Use: *appendfile*	Type: *time*	Default: *3s*

This specifies the time to wait between attempts to lock the file. See below for details of locking.

lock_retries	Use: *appendfile*	Type: *integer*	Default: *10*

This specifies the maximum number of attempts to lock the file. A value of zero is treated as 1. See below for details of locking.

lockfile_mode	Use: *appendfile*	Type: *octal integer*	Default: *0600*

This specifies the mode of the created lock file, when a lock file is being used (see **use_lockfile** and **use_mbx_lock**).

lockfile_timeout	Use: *appendfile*	Type: *time*	Default: *30m*

When a lock file is being used (see **use_lockfile**), 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.

mailbox_filecount	Use: *appendfile*	Type: *string*†	Default: *unset*

If this option is set, it is expanded, and the result is taken as the current number of files in the mailbox. It must be a decimal number, optionally followed by K or M. This provides a way of obtaining this information from an external source that maintains the data.

mailbox_size	Use: *appendfile*	Type: *string*†	Default: *unset*

If this option is set, it is expanded, and the result is taken as the current size the mailbox. It must be a decimal number, optionally followed by K or M. This provides a way of obtaining this information from an external source that maintains the data. This is likely to be helpful for maildir deliveries where it is computationally expensive to compute the size of a mailbox.

maildir_format	Use: *appendfile*	Type: *boolean*	Default: *false*

If this option is set with the **directory** option, the delivery is into a new file, in the "maildir" format that is used by other mail software. When the transport is activated directly from a *redirect* router (for example, the *address_file* transport in the default configuration), setting **maildir_format** causes the path received from the router to be treated as a directory, whether or not it ends with `/`. This option is available only if SUPPORT_MAILDIR is present in *Local/Makefile*. See section 26.5 below for further details.

maildir_quota_directory_regex	Use: *appendfile*	Type: *string*	Default: *See below*

This option is relevant only when **maildir_use_size_file** is set. It defines a regular expression for specifying directories, relative to the quota directory (see **quota_directory**), that should be included in the quota calculation. The default value is:

```
maildir_quota_directory_regex = ^(?:cur|new|\..*)$
```

This includes the *cur* and *new* directories, and any maildir++ folders (directories whose names begin with a dot). If you want to exclude the *Trash* folder from the count (as some sites do), you need to change this setting to

```
maildir_quota_directory_regex = ^(?:cur|new|\.(?!Trash).*)$
```

This uses a negative lookahead in the regular expression to exclude the directory whose name is *.Trash*. When a directory is excluded from quota calculations, quota processing is bypassed for any messages that are delivered directly into that directory.

maildir_retries	Use: *appendfile*	Type: *integer*	Default: *10*

This option specifies the number of times to retry when writing a file in "maildir" format. See section 26.5 below.

maildir_tag	Use: *appendfile*	Type: *string*†	Default: *unset*

This option applies only to deliveries in maildir format, and is described in section 26.5 below.

maildir_use_size_file	Use: *appendfile*	Type: *boolean*	Default: *false*

Setting this option true enables support for *maildirsize* files. Exim creates a *maildirsize* file in a maildir if one does not exist, taking the quota from the **quota** option of the transport. If **quota** is unset, the value is zero. See **maildir_quota_directory_regex** above and section 26.5 below for further details.

maildirfolder_create_regex	Use: *appendfile*	Type: *string*	Default: *unset*

The value of this option is a regular expression. If it is unset, it has no effect. Otherwise, before a maildir delivery takes place, the pattern is matched against the name of the maildir directory, that is, the directory containing the *new* and *tmp* subdirectories that will be used for the delivery. If there is a match, Exim checks for the existence of a file called *maildirfolder* in the directory, and creates it if it does not exist. See section 26.5 for more details.

mailstore_format	Use: *appendfile*	Type: *boolean*	Default: *false*

If this option is set with the **directory** option, the delivery is into two new files in "mailstore" format. The option is available only if SUPPORT_MAILSTORE is present in *Local/Makefile*. See section 26.4 below for further details.

mailstore_prefix	Use: *appendfile*	Type: *string*†	Default: *unset*

This option applies only to deliveries in mailstore format, and is described in section 26.4 below.

mailstore_suffix	Use: *appendfile*	Type: *string*†	Default: *unset*

This option applies only to deliveries in mailstore format, and is described in section 26.4 below.

mbx_format	Use: *appendfile*	Type: *boolean*	Default: *false*

This option is available only if Exim has been compiled with SUPPORT_MBX set in *Local/Makefile*. If **mbx_format** is set with the **file** option, the message is appended to the mailbox file in MBX format instead of traditional Unix format. This format is supported by Pine4 and its associated IMAP and POP daemons, by means of the *c-client* library that they all use.

Note: The **message_prefix** and **message_suffix** options are not automatically changed by the use of **mbx_format**. They should normally be set empty when using MBX format, so this option almost always appears in this combination:

```
mbx_format = true
message_prefix =
message_suffix =
```

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 specify the other kinds of locking with **mbx_format**, but **use_fcntl_lock** and **use_mbx_lock** are mutually exclusive. MBX locking interworks with *c-client*, 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 the standard version of *c-client*, because as long as it has a mailbox open (this means for the whole of a Pine or IMAP session), Exim will not be able to append messages to it.

message_prefix	Use: *appendfile*	Type: *string*†	Default: *see below*

The string specified here is expanded and output at the start of every message. The default is unset unless **file** is specified and **use_bsmtp** is not set, in which case it is:

```
message_prefix = "From ${if def:return_path{$return_path}\
  {MAILER-DAEMON}} $tod_bsdinbox\n"
```

Note: If you set **use_crlf** true, you must change any occurrences of `\n` to `\r\n` in **message_prefix**.

message_suffix	Use: *appendfile*	Type: *string*†	Default: *see below*

The string specified here is expanded and output at the end of every message. The default is unset unless **file** is specified and **use_bsmtp** is not set, in which case it is a single newline character. The suffix can be suppressed by setting

```
message_suffix =
```

Note: If you set **use_crlf** true, you must change any occurrences of `\n` to `\r\n` in **message_suffix**.

mode	Use: *appendfile*	Type: *octal integer*	Default: *0600*

If the output 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	Use: *appendfile*	Type: *boolean*	Default: *true*

This option applies in the case 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 deferred ("mailbox has the wrong mode"); otherwise Exim continues with the delivery attempt, using the existing mode of the file.

notify_comsat	Use: *appendfile*	Type: *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	Use: *appendfile*	Type: *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 when 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 sub-directories) have to be individually inspected and their sizes summed. (See **quota_size_regex** and **maildir_use_size_file** for ways to avoid this in environments where users have no shell access to their mailboxes).

As there is no interlock against two simultaneous deliveries into a multi-file mailbox, it is possible for the quota to be overrun in this case. For single-file mailboxes, of course, an interlock is a necessity.

A file's size is taken as its *used* value. Because of blocking effects, this may be a lot less than the actual amount of disk space allocated to the file. If the sizes of a number of files are being added up, the rounding effect can become quite noticeable, especially on systems that have large block sizes. Nevertheless, it seems best to stick to the *used* figure, because this is the obvious value which users understand most easily.

The value of the option is expanded, and must then be a numerical value (decimal point allowed), optionally followed by one of the letters K, M, or G, for kilobytes, megabytes, or gigabytes. If Exim is running on a system with large file support (Linux and FreeBSD have this), mailboxes larger than 2G can be handled.

Note: A value of zero is interpreted as "no quota".

The expansion happens while Exim is running as root, before it changes uid for the delivery. This means that files that are inaccessible to the end user can be used to hold quota values that are looked up in the expansion. When delivery fails because this quota is exceeded, the handling of the error is as for system quota failures.

By default, Exim's quota checking mimics system quotas, and restricts the mailbox to the specified maximum size, though the value is not accurate to the last byte, owing to separator lines and

additional headers that may get added during message delivery. When a mailbox is nearly full, large messages may get refused even though small ones are accepted, because the size of the current message is added to the quota when the check is made. This behaviour can be changed by setting **quota_is_inclusive** false. When this is done, the check for exceeding the quota does not include the current message. Thus, deliveries continue until the quota has been exceeded; thereafter, no further messages are delivered. See also **quota_warn_threshold**.

quota_directory	Use: *appendfile*	Type: *string*†	Default: *unset*

This option defines the directory to check for quota purposes when delivering into individual files. The default is the delivery directory, or, if a file called *maildirfolder* exists in a maildir directory, the parent of the delivery directory.

quota_filecount	Use: *appendfile*	Type: *string*†	Default: *0*

This option applies when the **directory** option is set. It limits the total number of files in the directory (compare the inode limit in system quotas). It can only be used if **quota** is also set. The value is expanded; an expansion failure causes delivery to be deferred. A value of zero is interpreted as "no quota".

quota_is_inclusive	Use: *appendfile*	Type: *boolean*	Default: *true*

See **quota** above.

quota_size_regex	Use: *appendfile*	Type: *string*	Default: *unset*

This option applies when one of the delivery modes that writes a separate file for each message is being used. When Exim wants to find the size of one of these files in order to test the quota, it first checks **quota_size_regex**. If this is set to a regular expression that matches the file name, and it captures one string, that string is interpreted as a representation of the file's size. The value of **quota_size_regex** is not expanded.

This feature is useful only when users have no shell access to their mailboxes – otherwise they could defeat the quota simply by renaming the files. This facility can be used with maildir deliveries, by setting **maildir_tag** to add the file length to the file name. For example:

```
maildir_tag = ,S=$message_size
quota_size_regex = ,S=(\d+)
```

An alternative to *$message_size* is *$message_linecount*, which contains the number of lines in the message.

The regular expression should not assume that the length is at the end of the file name (even though **maildir_tag** puts it there) because maildir MUAs sometimes add other information onto the ends of message file names.

quota_warn_message	Use: *appendfile*	Type: *string*†	Default: *see below*

See below for the use of this option. If it is not set when **quota_warn_threshold** is set, it defaults to

```
quota_warn_message = "\
  To: $local_part@$domain\n\
  Subject: Your mailbox\n\n\
  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"
```

quota_warn_threshold	Use: *appendfile*	Type: *string*†	Default: *0*

This option is expanded in the same way as **quota** (see above). 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. If **quota** is also set, the threshold may be specified as a percentage of it by following the value with a percent sign. For example:

```
quota = 10M
quota_warn_threshold = 75%
```

If **quota** is not set, a setting of **quota_warn_threshold** that ends with a percent sign is ignored.

The warning message itself is specified by the **quota_warn_message** option, and it must start with a *To:* header line containing the recipient(s) of the warning message. These do not necessarily have to include the recipient(s) of the original message. A *Subject:* line should also normally be supplied. You can include any other header lines that you want. If you do not include a *From:* line, the default is:

```
From: Mail Delivery System <mailer-daemon@$qualify_domain_sender>
```

If you supply a *Reply-To:* line, it overrides the global **errors_reply_to** option.

The **quota** option does not have to be set in order to use this option; they are independent of one another except when the threshold is specified as a percentage.

use_bsmtp	Use: *appendfile*	Type: *boolean*	Default: *false*

If this option is set true, *appendfile* writes messages in "batch SMTP" format, with the envelope sender and recipient(s) included as SMTP commands. If you want to include a leading HELO command with such messages, you can do so by setting the **message_prefix** option. See section 45.10 for details of batch SMTP.

use_crlf	Use: *appendfile*	Type: *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. In the case of batched SMTP, the byte sequence written to the file is then an exact image of what would be sent down a real SMTP connection.

Note: The contents of the **message_prefix** and **message_suffix** options (which are used to supply the traditional "From " and blank line separators in Berkeley-style mailboxes) are written verbatim, so must contain their own carriage return characters if these are needed. In cases where these options have non-empty defaults, the values end with a single linefeed, so they must be changed to end with `\r\n` if **use_crlf** is set.

use_fcntl_lock	Use: *appendfile*	Type: *boolean*	Default: *see below*

This option controls the use of the *fcntl()* function to lock a file for exclusive use when a message is being appended. It is set by default unless **use_flock_lock** is set. Otherwise, it should be turned off only if you know that all your MUAs use lock file locking. When both **use_fcntl_lock** and **use_flock_lock** are unset, **use_lockfile** must be set.

use_flock_lock	Use: *appendfile*	Type: *boolean*	Default: *false*

This option is provided to support the use of *flock()* for file locking, for the few situations where it is needed. Most modern operating systems support *fcntl()* and *lockf()* locking, and these two functions interwork with each other. Exim uses *fcntl()* locking by default.

This option is required only if you are using an operating system where *flock()* is used by programs that access mailboxes (typically MUAs), and where *flock()* does not correctly interwork with *fcntl()*. You can use both *fcntl()* and *flock()* locking simultaneously if you want.

Not all operating systems provide *flock()*. Some versions of Solaris do not have it (and some, I think, provide a not quite right version built on top of *lockf()*). If the OS does not have *flock()*, Exim will be built without the ability to use it, and any attempt to do so will cause a configuration error.

Warning: *flock()* locks do not work on NFS files (unless *flock()* is just being mapped onto *fcntl()* by the OS).

use_lockfile	Use: *appendfile*	Type: *boolean*	Default: *see below*

If this option is turned off, Exim does not attempt to create a lock file when appending to a mailbox file. In this situation, the only locking is by *fcntl()*. You should only turn **use_lockfile** off if you are absolutely sure that every MUA that is ever going to look at your users' mailboxes uses *fcntl()* rather than a lock file, and even then only when you are not delivering over NFS from more than one host.

In order to append to an NFS file safely from more than one host, it is necessary to take out a lock *before* opening the file, and the lock file achieves this. Otherwise, even with *fcntl()* locking, there is a risk of file corruption.

The **use_lockfile** option is set by default unless **use_mbx_lock** is set. It is not possible to turn both **use_lockfile** and **use_fcntl_lock** off, except when **mbx_format** is set.

use_mbx_lock	Use: *appendfile*	Type: *boolean*	Default: *see below*

This option is available only if Exim has been compiled with SUPPORT_MBX set in *Local/Makefile*. 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 Pine and the IMAP4 and POP daemons that come with it (see the discussion below). The rules allow for shared access to the mailbox. However, this kind of locking does not work when the mailbox is NFS mounted.

You can set **use_mbx_lock** with either (or both) of **use_fcntl_lock** and **use_flock_lock** to control what kind of locking is used in implementing the MBX locking rules. The default is to use *fcntl()* if **use_mbx_lock** is set without **use_fcntl_lock** or **use_flock_lock**.

26.3 Operational details for appending

Before appending to a file, the following preparations are made:

- If the name of the file is */dev/null*, no action is taken, and a success return is given.
- If any directories on the file's path are missing, Exim creates them if the **create_directory** option is set. A created directory's mode is given by the **directory_mode** option.
- If **file_format** is set, the format of an existing file is checked. If this indicates that a different transport should be used, control is passed to that transport.
- If **use_lockfile** is set, a lock file is built in a way that will work reliably over NFS, as follows:

  (1) Create a "hitching post" file whose name is that of the lock file with the current time, primary host name, and process id added, by opening for writing as a new file. If this fails with an access error, delivery is deferred.

  (2) Close the hitching post file, and hard link it to the lock file name.

  (3) If the call to *link()* succeeds, creation of the lock file has succeeded. Unlink the hitching post name.

  (4) Otherwise, use *stat()* to get information about the hitching post file, and then unlink hitching post name. If the number of links is exactly two, creation of the lock file succeeded but

something (for example, an NFS server crash and restart) caused this fact not to be communicated to the *link()* call.

(5) If creation of the lock file failed, wait for **lock_interval** and try again, up to **lock_retries** times. However, since any program that writes to a mailbox should complete its task very quickly, it is reasonable to time out old lock files that are normally the result of user agent and system crashes. If an existing lock file is older than **lockfile_timeout** Exim attempts to unlink it before trying again.

- A call is made to *lstat()* to discover whether the main file exists, and if so, what its characteristics are. If *lstat()* fails for any reason other than non-existence, delivery is deferred.
- If the file does exist and is a symbolic link, delivery is deferred, unless the **allow_symlink** option is set, in which case the ownership of the link is checked, and then *stat()* is called to find out about the real file, which is then subjected to the checks below. The check on the top-level link ownership prevents one user 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.
- If the file already exists but is not a regular file, or if the file's owner and group (if the group is being checked – see **check_group** above) are different from the user and group under which the delivery is running, delivery is deferred.
- If the file's permissions are more generous than specified, they are reduced. If they are insufficient, delivery is deferred, unless **mode_fail_narrower** is set false, in which case the delivery is tried using the existing permissions.
- The file's inode number is saved, and the file is then opened for appending. If this fails because the file has vanished, *appendfile* behaves as if it hadn't existed (see below). For any other failures, delivery is deferred.
- If the file is opened successfully, check that the inode number hasn't changed, that it is still a regular file, and that the owner and permissions have not changed. If anything is wrong, defer delivery and freeze the message.
- If the file did not exist originally, defer delivery if the **file_must_exist** option is set. Otherwise, check that the file is being created in a permitted directory if the **create_file** option is set (deferring on failure), and then open for writing as a new file, with the O_EXCL and O_CREAT options, except when dealing with a symbolic link (the **allow_symlink** option must be set). In this case, which can happen if the link points to a non-existent file, the file is opened for writing using O_CREAT but not O_EXCL, because that prevents link following.
- If opening fails because the file exists, obey the tests given above for existing files. However, to avoid looping in a situation where the file is being continuously created and destroyed, the exists/not-exists loop is broken after 10 repetitions, and the message is then frozen.
- If opening fails with any other error, defer delivery.
- Once the file is open, unless both **use_fcntl_lock** and **use_flock_lock** are false, it is locked using *fcntl()* or *flock()* or both. If **use_mbx_lock** is false, an exclusive lock is requested in each case. However, if **use_mbx_lock** is true, Exim takes out a shared lock on the open file, and an exclusive lock on the file whose name is

  ```
  /tmp/.<device-number>.<inode-number>
  ```

  using the device and inode numbers of the open mailbox file, in accordance with the MBX locking rules. This file is created with a mode that is specified by the **lockfile_mode** option.

  If Exim fails to lock the file, there are two possible courses of action, depending on the value of the locking timeout. This is obtained from **lock_fcntl_timeout** or **lock_flock_timeout**, as appropriate.

  If the timeout value is zero, the file is closed, Exim waits for **lock_interval**, and then goes back and re-opens the file as above and tries to lock it again. This happens up to **lock_retries** times, after which the delivery is deferred.

If the timeout has a value greater than zero, blocking calls to *fcntl()* or *flock()* are used (with the given timeout), so there has already been some waiting involved by the time locking fails. Nevertheless, Exim does not give up immediately. It retries up to

```
(lock_retries * lock_interval) / <timeout>
```

times (rounded up).

At the end of delivery, Exim closes the file (which releases the *fcntl()* and/or *flock()* locks) and then deletes the lock file if one was created.

26.4 Operational details for delivery to a new file

When the **directory** option is set instead of **file**, each message is delivered into a newly-created file or set of files. When *appendfile* is activated directly from a *redirect* router, neither **file** nor **directory** is normally set, because the path for delivery is supplied by the router. (See for example, the *address_file* transport in the default configuration.) In this case, delivery is to a new file if either the path name ends in `/`, or the **maildir_format** or **mailstore_format** option is set.

No locking is required while writing the message to a new file, so the various locking options of the transport are ignored. The "From" line that by default separates messages in a single file is not normally needed, nor is the escaping of message lines that start with "From", and there is no need to ensure a newline at the end of each message. Consequently, the default values for **check_string**, **message_prefix**, and **message_suffix** are all unset when any of **directory**, **maildir_format**, or **mailstore_format** is set.

If Exim is required to check a **quota** setting, it adds up the sizes of all the files in the delivery directory by default. However, you can specify a different directory by setting **quota_directory**. Also, for maildir deliveries (see below) the *maildirfolder* convention is honoured.

There are three different ways in which delivery to individual files can be done, controlled by the settings of the **maildir_format** and **mailstore_format** options. Note that code to support maildir or mailstore formats is not included in the binary unless SUPPORT_MAILDIR or SUPPORT_MAILSTORE, respectively, is set in *Local/Makefile*.

In all three cases an attempt is made to create the directory and any necessary sub-directories if they do not exist, provided that the **create_directory** option is set (the default). The location of a created directory can be constrained by setting **create_file**. A created directory's mode is given by the **directory_mode** option. If creation fails, or if the **create_directory** option is not set when creation is required, delivery is deferred.

26.5 Maildir delivery

If the **maildir_format** option is true, Exim delivers each message by writing it to a file whose name is *tmp/<stime>.H<mtime>P<pid>.<host>* in the directory that is defined by the **directory** option (the "delivery directory"). If the delivery is successful, the file is renamed into the *new* subdirectory.

In the file name, *<stime>* is the current time of day in seconds, and *<mtime>* is the microsecond fraction of the time. After a maildir delivery, Exim checks that the time-of-day clock has moved on by at least one microsecond before terminating the delivery process. This guarantees uniqueness for the file name. However, as a precaution, Exim calls *stat()* for the file before opening it. If any response other than ENOENT (does not exist) is given, Exim waits 2 seconds and tries again, up to **maildir_retries** times.

Before Exim carries out a maildir delivery, it ensures that subdirectories called *new*, *cur*, and *tmp* exist in the delivery directory. If they do not exist, Exim tries to create them and any superior directories in their path, subject to the **create_directory** and **create_file** options. If the **maildirfolder_create_regex** option is set, and the regular expression it contains matches the delivery directory, Exim also ensures that a file called *maildirfolder* exists in the delivery directory. If a missing directory or *maildirfolder* file cannot be created, delivery is deferred.

These features make it possible to use Exim to create all the necessary files and directories in a maildir mailbox, including subdirectories for maildir++ folders. Consider this example:

```
maildir_format = true
directory = /var/mail/$local_part\
           ${if eq{$local_part_suffix}{}{}\
           {/.${substr_1:$local_part_suffix}}}
maildirfolder_create_regex = /\.[^/]+$
```

If *$local_part_suffix* is empty (there was no suffix for the local part), delivery is into a toplevel maildir with a name like */var/mail/pimbo* (for the user called *pimbo*). The pattern in **maildirfolder_create_regex** does not match this name, so Exim will not look for or create the file */var/mail/pimbo/maildirfolder*, though it will create */var/mail/pimbo/{cur,new,tmp}* if necessary.

However, if *$local_part_suffix* contains `-eximusers` (for example), delivery is into the maildir++ folder */var/mail/pimbo/.eximusers*, which does match **maildirfolder_create_regex**. In this case, Exim will create */var/mail/pimbo/.eximusers/maildirfolder* as well as the three maildir directories */var/mail/pimbo/.eximusers/{cur,new,tmp}*.

Warning: Take care when setting **maildirfolder_create_regex** that it does not inadvertently match the toplevel maildir directory, because a *maildirfolder* file at top level would completely break quota calculations.

If Exim is required to check a **quota** setting before a maildir delivery, and **quota_directory** is not set, it looks for a file called *maildirfolder* in the maildir directory (alongside *new*, *cur*, *tmp*). If this exists, Exim assumes the directory is a maildir++ folder directory, which is one level down from the user's top level mailbox directory. This causes it to start at the parent directory instead of the current directory when calculating the amount of space used.

One problem with delivering into a multi-file mailbox is that it is computationally expensive to compute the size of the mailbox for quota checking. Various approaches have been taken to reduce the amount of work needed. The next two sections describe two of them. A third alternative is to use some external process for maintaining the size data, and use the expansion of the **mailbox_size** option as a way of importing it into Exim.

26.6 Using tags to record message sizes

If **maildir_tag** is set, the string is expanded for each delivery. When the maildir file is renamed into the *new* sub-directory, the tag is added to its name. However, if adding the tag takes the length of the name to the point where the test *stat()* call fails with ENAMETOOLONG, the tag is dropped and the maildir file is created with no tag.

Tags can be used to encode the size of files in their names; see **quota_size_regex** above for an example. The expansion of **maildir_tag** happens after the message has been written. The value of the *$message_size* variable is set to the number of bytes actually written. If the expansion is forced to fail, the tag is ignored, but a non-forced failure causes delivery to be deferred. The expanded tag may contain any printing characters except "/". Non-printing characters in the string are ignored; if the resulting string is empty, it is ignored. If it starts with an alphanumeric character, a leading colon is inserted.

26.7 Using a maildirsize file

If **maildir_use_size_file** is true, Exim implements the maildir++ rules for storing quota and message size information in a file called *maildirsize* within the toplevel maildir directory. If this file does not exist, Exim creates it, setting the quota from the **quota** option of the transport. If the maildir directory itself does not exist, it is created before any attempt to write a *maildirsize* file.

The *maildirsize* file is used to hold information about the sizes of messages in the maildir, thus speeding up quota calculations. The quota value in the file is just a cache; if the quota is changed in the transport, the new value overrides the cached value when the next message is delivered. The cache is maintained for the benefit of other programs that access the maildir and need to know the quota.

If the **quota** option in the transport is unset or zero, the *maildirsize* file is maintained (with a zero quota setting), but no quota is imposed.

A regular expression is available for controlling which directories in the maildir participate in quota calculations when a *maildirsizefile* is in use. See the description of the **maildir_quota_directory_regex** option above for details.

26.8 Mailstore delivery

If the **mailstore_format** option is true, each message is written as two files in the given directory. A unique base name is constructed from the message id and the current delivery process, and the files that are written use this base name plus the suffixes *.env* and *.msg*. The *.env* file contains the message's envelope, and the *.msg* file contains the message itself. The base name is placed in the variable *$mailstore_basename*.

During delivery, the envelope is first written to a file with the suffix *.tmp*. The *.msg* file is then written, and when it is complete, the *.tmp* file is renamed as the *.env* file. Programs that access messages in mailstore format should wait for the presence of both a *.msg* and a *.env* file before accessing either of them. An alternative approach is to wait for the absence of a *.tmp* file.

The envelope file starts with any text defined by the **mailstore_prefix** option, expanded and terminated by a newline if there isn't one. Then follows the sender address on one line, then all the recipient addresses, one per line. There can be more than one recipient only if the **batch_max** option is set greater than one. Finally, **mailstore_suffix** is expanded and the result appended to the file, followed by a newline if it does not end with one.

If expansion of **mailstore_prefix** or **mailstore_suffix** ends with a forced failure, it is ignored. Other expansion errors are treated as serious configuration errors, and delivery is deferred. The variable *$mailstore_basename* is available for use during these expansions.

26.9 Non-special new file delivery

If neither **maildir_format** nor **mailstore_format** is set, a single new file is created directly in the named directory. For example, when delivering messages into files in batched SMTP format for later delivery to some host (see section 45.10), a setting such as

```
directory = /var/bsmtp/$host
```

might be used. A message is written to a file with a temporary name, which is then renamed when the delivery is complete. The final name is obtained by expanding the contents of the **directory_file** option.

27. The autoreply transport

The *autoreply* transport is not a true transport in that it does not cause the message to be transmitted. Instead, it generates a new mail message as an automatic reply to the incoming message. *References:* and *Auto-Submitted:* header lines are included. These are constructed according to the rules in RFCs 2822 and 3834, respectively.

If the router that passes the message to this transport does not have the **unseen** option set, the original message (for the current recipient) is not delivered anywhere. However, when the **unseen** option is set on the router that passes the message to this transport, routing of the address continues, so another router can set up a normal message delivery.

The *autoreply* transport is usually run as the result of mail filtering, a "vacation" message being the standard example. However, it can also be run directly from a router like any other transport. To reduce the possibility of message cascades, messages created by the *autoreply* transport always have empty envelope sender addresses, like bounce messages.

The parameters of the message to be sent can be specified in the configuration by options described below. However, these are used only when the address passed to the transport does not contain its own reply information. When the transport is run as a consequence of a **mail** or **vacation** command in a filter file, the parameters of the message are supplied by the filter, and passed with the address. The transport's options that define the message are then ignored (so they are not usually set in this case). The message is specified entirely by the filter or by the transport; it is never built from a mixture of options. However, the **file_optional**, **mode**, and **return_message** options apply in all cases.

Autoreply is implemented as a local transport. When used as a result of a command in a user's filter file, *autoreply* normally runs under the uid and gid of the user, and with appropriate current and home directories (see chapter 23).

There is a subtle difference between routing a message to a *pipe* transport that generates some text to be returned to the sender, and routing 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.

Non-printing characters are not permitted in the header lines generated for the message that *autoreply* creates, with the exception of newlines that are immediately followed by white space. If any non-printing characters are found, the transport defers. Whether characters with the top bit set count as printing characters or not is controlled by the **print_topbitchars** global option.

If any of the generic options for manipulating headers (for example, **headers_add**) are set on an *autoreply* transport, they apply 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.

If the *autoreply* transport receives return code 2 from Exim when it submits the message, indicating that there were no recipients, it does not treat this as an error. This means that autoreplies sent to *$sender_address* when this is empty (because the incoming message is a bounce message) do not cause problems. They are just discarded.

27.1 Private options for autoreply

bcc	Use: *autoreply*	Type: *string*†	Default: *unset*

This specifies the addresses that are to receive "blind carbon copies" of the message when the message is specified by the transport.

cc	Use: *autoreply*	Type: *string*†	Default: *unset*

This specifies recipients of the message and the contents of the *Cc:* header when the message is specified by the transport.

file	Use: *autoreply*	Type: *string*†	Default: *unset*

The contents of the file are sent as the body of the message when the message is specified by the transport. If both **file** and **text** are set, the text string comes first.

file_expand	Use: *autoreply*	Type: *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	Use: *autoreply*	Type: *boolean*	Default: *false*

If this option is true, no error is generated if the file named by the **file** option or passed with the address does not exist or cannot be read.

from	Use: *autoreply*	Type: *string*†	Default: *unset*

This specifies the contents of the *From:* header when the message is specified by the transport.

headers	Use: *autoreply*	Type: *string*†	Default: *unset*

This specifies additional RFC 2822 headers that are to be added to the message when the message is specified by the transport. Several can be given by using "\n" to separate them. There is no check on the format.

log	Use: *autoreply*	Type: *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.

mode	Use: *autoreply*	Type: *octal integer*	Default: *0600*

If either the log file or the "once" file has to be created, this mode is used.

never_mail	Use: *autoreply*	Type: *address list*†	Default: *unset*

If any run of the transport creates a message with a recipient that matches any item in the list, that recipient is quietly discarded. If all recipients are discarded, no message is created. This applies both when the recipients are generated by a filter and when they are specified in the transport.

once	Use: *autoreply*	Type: *string*†	Default: *unset*

This option names a file or DBM database in which a record of each *To:* recipient is kept when the message is specified by the transport. **Note**: This does not apply to *Cc:* or *Bcc:* recipients.

If **once** is unset, or is set to an empty string, the message is always sent. By default, if **once** is set to a non-empty file name, the message is not sent if a potential recipient is already listed in the database. However, if the **once_repeat** option specifies a time greater than zero, the message is sent if that

much time has elapsed since a message was last sent to this recipient. A setting of zero time for **once_repeat** (the default) prevents a message from being sent a second time – in this case, zero means infinity.

If **once_file_size** is zero, a DBM database is used to remember recipients, and it is allowed to grow as large as necessary. If **once_file_size** is set greater than zero, it changes the way Exim implements the **once** option. Instead of using a DBM file to record every recipient it sends to, it uses a regular file, whose size will never get larger than the given value.

In the file, Exim keeps a linear list of recipient addresses and the times at which they were sent messages. If the file is full when a new address needs to be added, the oldest address is dropped. If **once_repeat** is not set, this means that a given recipient may receive multiple messages, but at unpredictable intervals that depend on the rate of turnover of addresses in the file. If **once_repeat** is set, it specifies a maximum time between repeats.

once_file_size	Use: *autoreply*	Type: *integer*	Default: *0*

See **once** above.

once_repeat	Use: *autoreply*	Type: *time*†	Default: *0s*

See **once** above. After expansion, the value of this option must be a valid time value.

reply_to	Use: *autoreply*	Type: *string*†	Default: *unset*

This specifies the contents of the *Reply-To:* header when the message is specified by the transport.

return_message	Use: *autoreply*	Type: *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** global configuration option.

subject	Use: *autoreply*	Type: *string*†	Default: *unset*

This specifies the contents of the *Subject:* header when the message is specified by the transport. It is tempting to quote the original subject in automatic responses. For example:

```
subject = Re: $h_subject:
```

There is a danger in doing this, however. It may allow a third party to subscribe your users to an opt-in mailing list, provided that the list accepts bounce messages as subscription confirmations. Well-managed lists require a non-bounce message to confirm a subscription, so the danger is relatively small.

text	Use: *autoreply*	Type: *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. If both **text** and **file** are set, the text comes first.

to	Use: *autoreply*	Type: *string*†	Default: *unset*

This specifies recipients of the message and the contents of the *To:* header when the message is specified by the transport.

28. The lmtp transport

The *lmtp* transport runs the LMTP protocol (RFC 2033) over a pipe to a specified command or by interacting with a Unix domain socket. This transport is something of a cross between the *pipe* and *smtp* transports. Exim also has support for using LMTP over TCP/IP; this is implemented as an option for the *smtp* transport. Because LMTP is expected to be of minority interest, the default build-time configure in *src/EDITME* has it commented out. You need to ensure that

```
TRANSPORT_LMTP=yes
```

is present in your *Local/Makefile* in order to have the *lmtp* transport included in the Exim binary. The private options of the *lmtp* transport are as follows:

batch_id	Use: *lmtp*	Type: *string*†	Default: *unset*

See the description of local delivery batching in chapter 25.

batch_max	Use: *lmtp*	Type: *integer*	Default: *1*

This limits the number of addresses that can be handled in a single delivery. Most LMTP servers can handle several addresses at once, so it is normally a good idea to increase this value. See the description of local delivery batching in chapter 25.

command	Use: *lmtp*	Type: *string*†	Default: *unset*

This option must be set if **socket** is not set. The string is a command which is run in a separate process. It is split up into a command name and list of arguments, each of which is separately expanded (so expansion cannot change the number of arguments). The command is run directly, not via a shell. The message is passed to the new process using the standard input and output to operate the LMTP protocol.

ignore_quota	Use: *lmtp*	Type: *boolean*	Default: *false*

If this option is set true, the string `IGNOREQUOTA` is added to RCPT commands, provided that the LMTP server has advertised support for IGNOREQUOTA in its response to the LHLO command.

socket	Use: *lmtp*	Type: *string*†	Default: *unset*

This option must be set if **command** is not set. The result of expansion must be the name of a Unix domain socket. The transport connects to the socket and delivers the message to it using the LMTP protocol.

timeout	Use: *lmtp*	Type: *time*	Default: *5m*

The transport is aborted if the created process or Unix domain socket does not respond to LMTP commands or message input within this timeout. Delivery is deferred, and will be tried again later. Here is an example of a typical LMTP transport:

```
lmtp:
  driver = lmtp
  command = /some/local/lmtp/delivery/program
  batch_max = 20
  user = exim
```

This delivers up to 20 addresses at a time, in a mixture of domains if necessary, running as the user *exim*.

29. The pipe transport

The *pipe* transport is used to deliver messages via a pipe to a command running in another process. One example is the use of *pipe* as a pseudo-remote transport for passing messages to some other delivery mechanism (such as UUCP). Another is the use by individual users to automatically process their incoming messages. The *pipe* transport can be used in one of the following ways:

- A router routes one address to a transport in the normal way, and the transport is configured as a *pipe* transport. In this case, *$local_part* contains the local part of the address (as usual), and the command that is run is specified by the **command** option on the transport.
- If the **batch_max** option is set greater than 1 (the default is 1), the transport can handle more than one address in a single run. In this case, when more than one address is routed to the transport, *$local_part* is not set (because it is not unique). However, the pseudo-variable *$pipe_addresses* (described in section 29.3 below) contains all the addresses that are routed to the transport.
- A router redirects an address directly to a pipe command (for example, from an alias or forward file). In this case, *$address_pipe* contains the text of the pipe command, and the **command** option on the transport is ignored. If only one address is being transported (**batch_max** is not greater than one, or only one address was redirected to this pipe command), *$local_part* contains the local part that was redirected.

The *pipe* transport is a non-interactive delivery method. Exim can also deliver messages over pipes using the LMTP interactive protocol. This is implemented by the *lmtp* transport.

In the case when *pipe* is run as a consequence of an entry in a local user's *.forward* file, the command runs under the uid and gid of that user. In other cases, the uid and gid have to be specified explicitly, either on the transport or on the router that handles the address. Current and "home" directories are also controllable. See chapter 23 for details of the local delivery environment and chapter 25 for a discussion of local delivery batching.

29.1 Concurrent delivery

If two messages arrive at almost the same time, and both are routed to a pipe delivery, the two pipe transports may be run concurrently. You must ensure that any pipe commands you set up are robust against this happening. If the commands write to a file, the **exim_lock** utility might be of use.

29.2 Returned status and data

If the command exits with a non-zero return code, the delivery is deemed to have failed, unless either the **ignore_status** option is set (in which case the return code is treated as zero), or the return code is one of those listed in the **temp_errors** option, which are interpreted as meaning "try again later". In this case, delivery is deferred. Details of a permanent failure are logged, but are not included in the bounce message, which merely contains "local delivery failed".

If the return code is greater than 128 and the command being run is a shell script, it normally means that the script was terminated by a signal whose value is the return code minus 128.

If Exim is unable to run the command (that is, if *execve()* fails), the return code is set to 127. This is the value that a shell returns if it is asked to run a non-existent command. The wording for the log line suggests that a non-existent command may be the problem.

The **return_output** option can affect the result of a pipe delivery. If it is set and the command produces any output on its standard output or standard error streams, the command is considered to have failed, even if it gave a zero return code or if **ignore_status** is set. The output from the command is included as part of the bounce message. The **return_fail_output** option is similar, except that output is returned only when the command exits with a failure return code, that is, a value other than zero or a code that matches **temp_errors**.

29.3 How the command is run

The command line is (by default) broken down into a command name and arguments by the *pipe* transport itself. The **allow_commands** and **restrict_to_path** options can be used to restrict the commands that may be run.

Unquoted arguments are delimited by white space. If an argument appears in double quotes, backslash is interpreted as an escape character in the usual way. If an argument appears in single quotes, no escaping is done.

String expansion is applied to the command line except when it comes from a traditional *.forward* file (commands from a filter file are expanded). The expansion is applied to each argument in turn rather than to the whole line. For this reason, any string expansion item that contains white space must be quoted so as to be contained within a single argument. A setting such as

```
command = /some/path ${if eq{$local_part}{postmaster}{xx}{yy}}
```

will not work, because the expansion item gets split between several arguments. You have to write

```
command = /some/path "${if eq{$local_part}{postmaster}{xx}{yy}}"
```

to ensure that it 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. However, this leads to problems if you want to generate multiple arguments (or the command name plus arguments) from a single expansion. In this situation, the simplest solution is to use a shell. For example:

```
command = /bin/sh -c ${lookup{$local_part}lsearch{/some/file}}
```

Special handling takes place when an argument consists of precisely 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 at that point *as a separate argument*. This avoids any problems with spaces or shell metacharacters, and is of use when a *pipe* transport is handling groups of addresses in a batch.

After splitting up into arguments and expansion, the resulting command is run in a subprocess directly from the transport, *not* under a shell. The message that is being delivered is supplied on the standard input, and the standard output and standard error are both connected to a single pipe that is read by Exim. The **max_output** option controls how much output the command may produce, and the **return_output** and **return_fail_output** options control what is done with it.

Not running the command under a shell (by default) lessens the security risks in cases when a command from a user's filter file is built out of data that was taken from an incoming message. If a shell is required, it can of course be explicitly specified as the command to be run. However, there are circumstances where existing commands (for example, in *.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*. The **restrict_to_path** option and the *$pipe_addresses* facility cannot be used with **use_shell**, and the whole mechanism is inherently less secure.

29.4 Environment variables

The environment variables listed below are set up when the command is invoked. This list is a compromise for maximum compatibility with other MTAs. Note that the **environment** option can be used to add additional variables to this environment.

`DOMAIN`	the domain of the address
`HOME`	the home directory, if set
`HOST`	the host name when called from a router (see below)
`LOCAL_PART`	see below
`LOCAL_PART_PREFIX`	see below
`LOCAL_PART_SUFFIX`	see below

LOGNAME	see below
MESSAGE_ID	Exim's local ID for the message
PATH	as specified by the **path** option below
QUALIFY_DOMAIN	the sender qualification domain
RECIPIENT	the complete recipient address
SENDER	the sender of the message (empty if a bounce)
SHELL	/bin/sh
TZ	the value of the **timezone** option, if set
USER	see below

When a *pipe* transport is called directly from (for example) an *accept* router, LOCAL_PART is set to the local part of the address. When it is called as a result of a forward or alias expansion, LOCAL_PART is set to the local part of the address that was expanded. In both cases, any affixes are removed from the local part, and made available in LOCAL_PART_PREFIX and LOCAL_PART_SUFFIX, respectively. 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 that associates hosts with an address, typically when using *pipe* as a pseudo-remote transport. HOST is set to the first host name specified by the router.

If the transport's generic **home_directory** option is set, its value is used for the HOME environment variable. Otherwise, a home directory may be set by the router's **transport_home_directory** option, which defaults to the user's home directory if **check_local_user** is set.

29.5 Private options for pipe

allow_commands	Use: *pipe*	Type: *string list*†	Default: *unset*

The string is expanded, and is then 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 still used for relative paths. If **restrict_to_path** is set with **allow_commands**, the command must either be in the **allow_commands** list, or a name without any slashes that is found on the path. In other words, if neither **allow_commands** nor **restrict_to_path** is set, there is no restriction on the command, but otherwise only commands that are permitted by one or the other are allowed. For example, if

```
allow_commands = /usr/bin/vacation
```

and **restrict_to_path** is not set, the only permitted command is */usr/bin/vacation*. The **allow_commands** option may not be set if **use_shell** is set.

batch_id	Use: *pipe*	Type: *string*†	Default: *unset*

See the description of local delivery batching in chapter 25.

batch_max	Use: *pipe*	Type: *integer*	Default: *1*

This limits the number of addresses that can be handled in a single delivery. See the description of local delivery batching in chapter 25.

check_string	Use: *pipe*	Type: *string*	Default: *unset*

As *pipe* 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**, provided both are set. The value of **check_string** is a literal string, not a regular expression, and the case of any letters it contains is significant. When **use_bsmtp** is set, the contents of **check_string** and **escape_string** are

forced to values that implement the SMTP escaping protocol. Any settings made in the configuration file are ignored.

command	Use: *pipe*	Type: *string*†	Default: *unset*

This option need not be set when *pipe* is being used to deliver to pipes obtained directly from address redirections. In other cases, the option must be set, to provide a command to be run. It need not yield an absolute path (see the **path** option below). The command is split up into separate arguments by Exim, and each argument is separately expanded, as described in section 29.3 above.

environment	Use: *pipe*	Type: *string*†	Default: *unset*

This option is used to add additional variables to the environment in which the command runs (see section 29.4 for the default list). Its value is a string which is expanded, and then interpreted as a colon-separated list of environment settings of the form <*name*>=<*value*>.

escape_string	Use: *pipe*	Type: *string*	Default: *unset*

See **check_string** above.

freeze_exec_fail	Use: *pipe*	Type: *boolean*	Default: *false*

Failure to exec 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 exec is treated specially, and causes the message to be frozen, whatever the setting of **ignore_status**.

ignore_status	Use: *pipe*	Type: *boolean*	Default: *false*

If this option is true, the status returned by the subprocess that is set up to run the command is ignored, and Exim behaves as if zero had been returned. Otherwise, a non-zero status or termination by signal causes an error return from the transport unless the status value is one of those listed in **temp_errors**; these cause the delivery to be deferred and tried again later.

Note: This option does not apply to timeouts, which do not return a status. See the **timeout_defer** option for how timeouts are handled.

log_defer_output	Use: *pipe*	Type: *boolean*	Default: *false*

If this option is set, and the status returned by the command is one of the codes listed in **temp_errors** (that is, delivery was deferred), and any output was produced, the first line of it is written to the main log.

log_fail_output	Use: *pipe*	Type: *boolean*	Default: *false*

If this option is set, and the command returns any output, and also ends with a return code that is neither zero nor one of the return codes listed in **temp_errors** (that is, the delivery failed), the first line of output is written to the main log. This option and **log_output** are mutually exclusive. Only one of them may be set.

log_output	Use: *pipe*	Type: *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. This option and **log_fail_output** are mutually exclusive. Only one of them may be set.

max_output	Use: *pipe*	Type: *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. This is intended as a safety measure to catch runaway processes. The limit is applied independently of the settings of the options that control what is done with such output (for example, **return_output**). Because of buffering effects, the amount of output may exceed the limit by a small amount before Exim notices.

message_prefix	Use: *pipe*	Type: *string*†	Default: *see below*

The string specified here is expanded and output at the start of every message. The default is unset if **use_bsmtp** is set. Otherwise it is

```
message_prefix = \
  From ${if def:return_path{$return_path}{MAILER-DAEMON}}\
  ${tod_bsdinbox}\n
```

This is required by the commonly used */usr/bin/vacation* program. However, it must *not* be present if delivery is to the Cyrus IMAP server, or to the **tmail** local delivery agent. The prefix can be suppressed by setting

```
message_prefix =
```

Note: If you set **use_crlf** true, you must change any occurrences of `\n` to `\r\n` in **message_prefix**.

message_suffix	Use: *pipe*	Type: *string*†	Default: *see below*

The string specified here is expanded and output at the end of every message. The default is unset if **use_bsmtp** is set. Otherwise it is a single newline. The suffix can be suppressed by setting

```
message_suffix =
```

Note: If you set **use_crlf** true, you must change any occurrences of `\n` to `\r\n` in **message_suffix**.

path	Use: *pipe*	Type: *string*	Default: *see below*

This option specifies the string that is set up in the PATH environment variable of the subprocess. The default is:

```
/bin:/usr/bin
```

If the **command** option does not yield an absolute path name, the command is sought in the PATH directories, in the usual way. **Warning**: This does not apply to a command specified as a transport filter.

pipe_as_creator	Use: *pipe*	Type: *boolean*	Default: *false*

If the generic **user** option is not set and this option is true, the delivery process is run under the uid that was in force when Exim was originally called to accept the message. If the group id is not otherwise set (via the generic **group** option), the gid that was in force when Exim was originally called to accept the message is used.

restrict_to_path	Use: *pipe*	Type: *boolean*	Default: *false*

When this option is set, any command name not listed in **allow_commands** must contain no slashes. The command is searched for only in the directories listed in the **path** option. This option is intended for use in the case when a pipe command has been generated from a user's *.forward* file. This is usually handled by a *pipe* transport called **address_pipe**.

return_fail_output	Use: *pipe*	Type: *boolean*	Default: *false*

If this option is true, and the command produced any output and ended with a return code other than zero or one of the codes listed in **temp_errors** (that is, the delivery failed), the output is returned in the bounce message. However, if the message has a null sender (that is, it is itself a bounce message), output from the command is discarded. This option and **return_output** are mutually exclusive. Only one of them may be set.

return_output	Use: *pipe*	Type: *boolean*	Default: *false*

If this option is true, and the command produced any output, the delivery is deemed to have failed whatever the return code from the command, and the output is returned in the bounce message. Otherwise, the output is just discarded. However, if the message has a null sender (that is, it is a bounce message), output from the command is always discarded, whatever the setting of this option. This option and **return_fail_output** are mutually exclusive. Only one of them may be set.

temp_errors	Use: *pipe*	Type: *string list*	Default: *see below*

This option contains either a colon-separated list of numbers, or a single asterisk. If **ignore_status** is false and **return_output** is not set, and the command exits with a non-zero return code, the failure is treated as temporary and the delivery is deferred if the return code matches one of the numbers, or if the setting is a single asterisk. Otherwise, non-zero return codes are treated as permanent errors. 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	Use: *pipe*	Type: *time*	Default: *1h*

If the command fails to complete within this time, it is killed. This normally causes the delivery to fail (but see **timeout_defer**). A zero time interval specifies no timeout. In order to ensure that any subprocesses 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 can be defeated if one of the processes starts a new process group.

timeout_defer	Use: *pipe*	Type: *boolean*	Default: *false*

A timeout in a *pipe* transport, either in the command that the transport runs, or in a transport filter that is associated with it, is by default treated as a hard error, and the delivery fails. However, if **timeout_defer** is set true, both kinds of timeout become temporary errors, causing the delivery to be deferred.

umask	Use: *pipe*	Type: *octal integer*	Default: *022*

This specifies the umask setting for the subprocess that runs the command.

use_bsmtp	Use: *pipe*	Type: *boolean*	Default: *false*

If this option is set true, the *pipe* transport writes messages in "batch SMTP" format, with the envelope sender and recipient(s) included as SMTP commands. If you want to include a leading HELO command with such messages, you can do so by setting the **message_prefix** option. See section 45.10 for details of batch SMTP.

use_classresources	Use: *pipe*	Type: *boolean*	Default: *false*

This option is available only when Exim is running on FreeBSD, NetBSD, or BSD/OS. If it is set true, the *setclassresources()* function is used to set resource limits when a *pipe* transport is run to perform a delivery. The limits for the uid under which the pipe is to run are obtained from the login class database.

use_crlf	Use: *pipe*	Type: *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. In the case of batched SMTP, the byte sequence written to the pipe is then an exact image of what would be sent down a real SMTP connection.

The contents of the **message_prefix** and **message_suffix** options are written verbatim, so must contain their own carriage return characters if these are needed. When **use_bsmtp** is not set, the default values for both **message_prefix** and **message_suffix** end with a single linefeed, so their values must be changed to end with `\r\n` if **use_crlf** is set.

use_shell	Use: *pipe*	Type: *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, as described in section 29.3. 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. The **allow_commands** and **restrict_to_path** options, and the `$pipe_addresses` facility are incompatible with **use_shell**. The command is expanded as a single string, and handed to */bin/sh* as data for its **-c** option.

29.6 Using an external local delivery agent

The *pipe* transport can be used to pass all messages that require local delivery to a separate local delivery agent such as **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 transport and router configuration for **procmail**:

```
# transport
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

# router
procmail:
  driver = accept
```

```
  check_local_user
  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*, but 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.

Note: The command that the pipe transport runs does *not* begin with

```
IFS=" "
```

as shown in some **procmail** documentation, because Exim does not by default use a shell to run pipe commands.

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

```
# transport
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
  message_prefix =
  message_suffix =

# router
local_user_cyrus:
  driver = accept
  check_local_user
  local_part_suffix = .*
  transport = local_delivery_cyrus
```

Note the unsetting of **message_prefix** and **message_suffix**, and the use of **return_output** to cause any text written by Cyrus to be returned to the sender.

30. The smtp transport

The *smtp* transport delivers messages over TCP/IP connections using the SMTP or LMTP protocol. The list of hosts to try can either be taken from the address that is being processed (having been set up by the router), or specified explicitly for the transport. Timeout and retry processing (see chapter 32) is applied to each IP address independently.

30.1 Multiple messages on a single connection

The sending of multiple messages over a single TCP/IP connection can arise in two ways:

- If a message contains more than **max_rcpt** (see below) addresses that are routed to the same host, more than one copy of the message has to be sent to that host. In this situation, multiple copies may be sent in a single run of the *smtp* transport over a single TCP/IP connection. (What Exim actually does when it has too many addresses to send in one message also depends on the value of the global **remote_max_parallel** option. Details are given in section 45.1.)
- 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 send multiple copies and possibly create yet another process.

For each copy sent over the same TCP/IP connection, a sequence counter is incremented, and if it ever gets to the value of **connection_max_messages**, no further messages are sent over that connection.

30.2 Use of the $host and $host_address variables

At the start of a run of the *smtp* transport, the values of *$host* and *$host_address* are the name and IP address of the first host on the host list passed by the router. However, when the transport is about to connect to a specific host, and while it is connected to that host, *$host* and *$host_address* are set to the values for that host. These are the values that are in force when the **helo_data**, **hosts_try_auth**, **interface**, **serialize_hosts**, and the various TLS options are expanded.

30.3 Use of $tls_cipher and $tls_peerdn

At the start of a run of the *smtp* transport, the values of *$tls_cipher* and *$tls_peerdn* are the values that were set when the message was received. These are the values that are used for options that are expanded before any SMTP connections are made. Just before each connection is made, these two variables are emptied. If TLS is subsequently started, they are set to the appropriate values for the outgoing connection, and these are the values that are in force when any authenticators are run and when the **authenticated_sender** option is expanded.

30.4 Private options for smtp

The private options of the *smtp* transport are as follows:

address_retry_include_sender	Use: *smtp*	Type: *boolean*	Default: *true*

When an address is delayed because of a 4*xx* response to a RCPT command, it is the combination of sender and recipient that is delayed in subsequent queue runs until the retry time is reached. You can delay the recipient without reference to the sender (which is what earlier versions of Exim did), by setting **address_retry_include_sender** false. However, this can lead to problems with servers that regularly issue 4*xx* responses to RCPT commands.

allow_localhost	Use: *smtp*	Type: *boolean*	Default: *false*

When a host specified in **hosts** or **fallback_hosts** (see below) turns out to be the local host, or is listed in **hosts_treat_as_local**, delivery is deferred by default. However, if **allow_localhost** is set, Exim 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).

authenticated_sender	Use: *smtp*	Type: *string*†	Default: *unset*

When Exim has authenticated as a client, or if **authenticated_sender_force** is true, this option sets a value for the AUTH= item on outgoing MAIL commands, overriding any existing authenticated sender value. If the string expansion is forced to fail, the option is ignored. Other expansion failures cause delivery to be deferred. If the result of expansion is an empty string, that is also ignored.

The expansion happens after the outgoing connection has been made and TLS started, if required. This means that the *$host*, *$host_address*, *$tls_cipher*, and *$tls_peerdn* variables are set according to the particular connection.

If the SMTP session is not authenticated, the expansion of **authenticated_sender** still happens (and can cause the delivery to be deferred if it fails), but no AUTH= item is added to MAIL commands unless **authenticated_sender_force** is true.

This option allows you to use the *smtp* transport in LMTP mode to deliver mail to Cyrus IMAP and provide the proper local part as the "authenticated sender", via a setting such as:

```
authenticated_sender = $local_part
```

This removes the need for IMAP subfolders to be assigned special ACLs to allow direct delivery to those subfolders.

Because of expected uses such as that just described for Cyrus (when no domain is involved), there is no checking on the syntax of the provided value.

authenticated_sender_force	Use: *smtp*	Type: *boolean*	Default: *false*

If this option is set true, the **authenticated_sender** option's value is used for the AUTH= item on outgoing MAIL commands, even if Exim has not authenticated as a client.

command_timeout	Use: *smtp*	Type: *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	Use: *smtp*	Type: *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 act. To have any effect, the value of this option must be less than the system timeout. However, it has been observed that on some systems there is no system timeout, which is why the default value for this option is 5 minutes, a value recommended by RFC 1123.

connection_max_messages	Use: *smtp*	Type: *integer*	Default: *500*

This controls the maximum number of separate message deliveries that are sent over a single TCP/IP connection. If the value is zero, there is no limit. For testing purposes, this value can be overridden by the **-oB** command line option.

data_timeout	Use: *smtp*	Type: *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. See also **final_timeout**.

delay_after_cutoff	Use: *smtp*	Type: *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.

In the default state, if the next retry time has not been reached for any of them, the address is bounced without trying any deliveries. In other words, Exim delays retrying an IP address after the final cutoff time until a new retry time is reached, and can therefore bounce an address without ever trying a delivery, when machines have been down for a long time. Some people are unhappy at this prospect, so...

If **delay_after_cutoff** is set false, Exim behaves differently. If all IP addresses 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, of if they all fail, the address is bounced. In other words, it does not delay when a new message arrives, but immediately tries those expired IP addresses that haven't been tried since the message arrived. If there is a continuous stream of messages for the dead hosts, unsetting **delay_after_cutoff** means that there will be many more attempts to deliver to them.

dns_qualify_single	Use: *smtp*	Type: *boolean*	Default: *true*

If the **hosts** or **fallback_hosts** option is being used, and the **gethostbyname** option is false, the RES_ DEFNAMES resolver option is set. See the **qualify_single** option in chapter 17 for more details.

dns_search_parents	Use: *smtp*	Type: *boolean*	Default: *false*

If the **hosts** or **fallback_hosts** option is being used, and the **gethostbyname** option is false, the RES_ DNSRCH resolver option is set. See the **search_parents** option in chapter 17 for more details.

fallback_hosts	Use: *smtp*	Type: *string list*	Default: *unset*

String expansion is not applied to this option. The argument must be a colon-separated list of host names or IP addresses, optionally also including port numbers, though the separator can be changed, as described in section 6.19. Each individual item in the list is the same as an item in a **route_list** setting for the *manualroute* router, as described in section 20.5.

Fallback hosts can also be specified on routers, which associate them with the addresses they process. As for the **hosts** option without **hosts_override**, **fallback_hosts** specified on the transport is used only if the address does not have its own associated fallback host list. Unlike **hosts**, a setting of **fallback_hosts** on an address is not overridden by **hosts_override**. However, **hosts_randomize** does apply to fallback host lists.

If Exim is unable to deliver to any of the hosts for a particular address, and the errors are not permanent rejections, the address is put on a separate transport queue with its host list replaced by the fallback hosts, unless the address was routed via MX records and the current host was in the original MX list. In that situation, the fallback host list is not used.

Once normal deliveries are complete, the fallback queue is delivered by re-running 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.

The resolution of the host names on the fallback list is controlled by the **gethostbyname** option, as for the **hosts** option. Fallback hosts apply both to cases when the host list comes with the address and when it is taken from **hosts**. This option provides a "use a smart host only if delivery fails" facility.

final_timeout	Use: *smtp*	Type: *time*	Default: *10m*

This is the timeout that applies while waiting for the response to the final line containing just "." that terminates a message. Its value must not be zero.

gethostbyname	Use: *smtp*	Type: *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()* (or *getipnodebyname()* when available) instead of using the DNS. Of course, that function may in fact use the DNS, but it may also consult other sources of information such as */etc/hosts*.

gnutls_require_kx	Use: *main*	Type: *string*	Default: *unset*

This option controls the key exchange mechanisms when GnuTLS is used in an Exim client. For details, see section 39.5.

gnutls_require_mac	Use: *main*	Type: *string*	Default: *unset*

This option controls the MAC algorithms when GnuTLS is used in an Exim client. For details, see section 39.5.

gnutls_require_protocols	Use: *main*	Type: *string*	Default: *unset*

This option controls the protocols when GnuTLS is used in an Exim client. For details, see section 39.5.

helo_data	Use: *smtp*	Type: *string*†	Default: *see below*

The value of this option is expanded after a connection to a another host has been set up. The result is used as the argument for the EHLO, HELO, or LHLO command that starts the outgoing SMTP or LMTP session. The default value of the option is:

```
$primary_hostname
```

During the expansion, the variables *$host* and *$host_address* are set to the identity of the remote host, and the variables *$sending_ip_address* and *$sending_port* are set to the local IP address and port number that are being used. These variables can be used to generate different values for different servers or different local IP addresses. For example, if you want the string that is used for **helo_data** to be obtained by a DNS lookup of the outgoing interface address, you could use this:

```
helo_data = ${lookup dnsdb{ptr=$sending_ip_address}{$value}\
  {$primary_hostname}}
```

The use of **helo_data** applies both to sending messages and when doing callouts.

hosts	Use: *smtp*	Type: *string list*†	Default: *unset*

Hosts are associated with an address by a router such as *dnslookup*, which finds the hosts by looking up the address domain in the DNS, or by *manualroute*, which has lists of hosts in its configuration.

However, email addresses can be passed to the *smtp* transport by any router, and not all of them can provide an associated list of hosts.

The **hosts** option specifies a list of hosts to be used if the address being processed does not have any hosts associated with it. The hosts specified by **hosts** are also used, whether or not the address has its own hosts, if **hosts_override** is set.

The string is first expanded, before being interpreted as a colon-separated list of host names or IP addresses, possibly including port numbers. The separator may be changed to something other than colon, as described in section 6.19. Each individual item in the list is the same as an item in a **route_list** setting for the *manualroute* router, as described in section 20.5. However, note that the `/MX` facility of the *manualroute* router is not available here.

If the expansion fails, delivery is deferred. Unless the failure was caused by the inability to complete a lookup, the error is logged to the panic log as well as the main log. Host names are looked up either by searching directly for address records in the DNS or by calling *gethostbyname()* (or *getipnodebyname()* when available), depending on the setting of the **gethostbyname** option. When Exim is compiled with IPv6 support, if a host that is looked up in the DNS has both IPv4 and IPv6 addresses, both types of address are used.

During delivery, the hosts are tried in order, subject to their retry status, unless **hosts_randomize** is set.

hosts_avoid_esmtp	Use: *smtp*	Type: *host list*†	Default: *unset*

This option is for use with broken hosts that announce ESMTP facilities (for example, PIPELINING) and then fail to implement them properly. When a host matches **hosts_avoid_esmtp**, Exim sends HELO rather than EHLO at the start of the SMTP session. This means that it cannot use any of the ESMTP facilities such as AUTH, PIPELINING, SIZE, and STARTTLS.

hosts_avoid_pipelining	Use: *smtp*	Type: *host list*†	Default: *unset*

Exim will not use the SMTP PIPELINING extension when delivering to any host that matches this list, even if the server host advertises PIPELINING support.

hosts_avoid_tls	Use: *smtp*	Type: *host list*†	Default: *unset*

Exim will not try to start a TLS session when delivering to any host that matches this list. See chapter 39 for details of TLS.

hosts_max_try	Use: *smtp*	Type: *integer*	Default: *5*

This option limits the number of IP addresses that are tried for any one delivery in cases where there are temporary delivery errors. Section 30.5 describes in detail how the value of this option is used.

hosts_max_try_hardlimit	Use: *smtp*	Type: *integer*	Default: *50*

This is an additional check on the maximum number of IP addresses that Exim tries for any one delivery. Section 30.5 describes its use and why it exists.

hosts_nopass_tls	Use: *smtp*	Type: *host list*†	Default: *unset*

For any host that matches this list, a connection on which a TLS session has been started will not be passed to a new delivery process for sending another message on the same connection. See section 39.10 for an explanation of when this might be needed.

hosts_override	Use: *smtp*	Type: *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 always used. This option does not apply to **fallback_hosts**.

hosts_randomize	Use: *smtp*	Type: *boolean*	Default: *false*

If this option is set, and either the list of hosts is taken from the **hosts** or the **fallback_hosts** option, or the hosts supplied by the router were not obtained from MX records (this includes fallback hosts from the router), and were not randomized by the router, the order of trying the hosts is randomized each time the transport runs. Randomizing the order of a host list can be used to do crude load sharing.

When **hosts_randomize** is true, a host list may be split into groups whose order is separately randomized. This makes it possible to set up MX-like behaviour. The boundaries between groups are indicated by an item that is just + in the host list. For example:

```
hosts = host1:host2:host3:+:host4:host5
```

The order of the first three hosts and the order of the last two hosts is randomized for each use, but the first three always end up before the last two. If **hosts_randomize** is not set, a + item in the list is ignored.

hosts_require_auth	Use: *smtp*	Type: *host list*†	Default: *unset*

This option provides a list of servers for which authentication must succeed before Exim will try to transfer a message. If authentication fails for servers which are not in this list, Exim tries to send unauthenticated. If authentication fails for one of these servers, delivery is deferred. This temporary error is detectable in the retry rules, so it can be turned into a hard failure if required. See also **hosts_try_auth**, and chapter 33 for details of authentication.

hosts_require_tls	Use: *smtp*	Type: *host list*†	Default: *unset*

Exim will insist on using a TLS session when delivering to any host that matches this list. See chapter 39 for details of TLS. **Note**: This option affects outgoing mail only. To insist on TLS for incoming messages, use an appropriate ACL.

hosts_try_auth	Use: *smtp*	Type: *host list*†	Default: *unset*

This option provides a list of servers to which, provided they announce authentication support, Exim will attempt to authenticate as a client when it connects. If authentication fails, Exim will try to transfer the message unauthenticated. See also **hosts_require_auth**, and chapter 33 for details of authentication.

interface	Use: *smtp*	Type: *string list*†	Default: *unset*

This option specifies which interface to bind to when making an outgoing SMTP call. The value is an IP address, not an interface name such as `eth0`. Do not confuse this with the interface address that was used when a message was received, which is in *$received_ip_address*, formerly known as *$interface_address*. The name was changed to minimize confusion with the outgoing interface address. There is no variable that contains an outgoing interface address because, unless it is set by this option, its value is unknown.

During the expansion of the **interface** option the variables *$host* and *$host_address* refer to the host to which a connection is about to be made during the expansion of the string. Forced expansion failure, or an empty string result causes the option to be ignored. Otherwise, after expansion, the

string must be a list of IP addresses, colon-separated by default, but the separator can be changed in the usual way. For example:

```
interface = <; 192.168.123.123 ; 3ffe:ffff:836f::fe86:a061
```

The first interface of the correct type (IPv4 or IPv6) is used for the outgoing connection. If none of them are the correct type, the option is ignored. If **interface** is not set, or is ignored, the system's IP functions choose which interface to use if the host has more than one.

keepalive	Use: *smtp*	Type: *boolean*	Default: *true*

This option controls the setting of SO_KEEPALIVE on outgoing TCP/IP socket connections. When set, it causes the kernel to probe idle connections periodically, by sending packets with "old" sequence numbers. The other end of the connection should send a acknowledgment if the connection is still okay or a reset if the connection has been aborted. The reason for doing this is that it has the beneficial effect of freeing up certain types of connection that can get stuck when the remote host is disconnected without tidying up the TCP/IP call properly. The keepalive mechanism takes several hours to detect unreachable hosts.

lmtp_ignore_quota	Use: *smtp*	Type: *boolean*	Default: *false*

If this option is set true when the **protocol** option is set to "lmtp", the string `IGNOREQUOTA` is added to RCPT commands, provided that the LMTP server has advertised support for IGNOREQUOTA in its response to the LHLO command.

max_rcpt	Use: *smtp*	Type: *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	Use: *smtp*	Type: *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. This is useful if you want to use *$domain* in an expansion for the transport, because it is set only when there is a single domain involved in a remote delivery.

port	Use: *smtp*	Type: *string*†	Default: *see below*

This option specifies the TCP/IP port on the server to which Exim connects. **Note:** Do not confuse this with the port that was used when a message was received, which is in *$received_port*, formerly known as *$interface_port*. The name was changed to minimize confusion with the outgoing port. There is no variable that contains an outgoing port.

If the value of this option begins with a digit it is taken as a port number; otherwise it is looked up using *getservbyname()*. The default value is normally "smtp", but if **protocol** is set to "lmtp", the default is "lmtp". If the expansion fails, or if a port number cannot be found, delivery is deferred.

protocol	Use: *smtp*	Type: *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 operates the LMTP protocol (RFC 2033) instead of SMTP. This protocol is sometimes used for local deliveries into closed message stores. Exim also has support for running LMTP over a pipe to a local process – see chapter 28.

retry_include_ip_address	Use: *smtp*	Type: *boolean*	Default: *true*

Exim normally includes both the host name and the IP address in the key it constructs for indexing retry data after a temporary delivery failure. This means that when one of several IP addresses for a host is failing, it gets tried periodically (controlled by the retry rules), but use of the other IP addresses is not affected.

However, in some dialup environments hosts are assigned a different IP address each time they connect. In this situation the use of the IP address as part of the retry key leads to undesirable behaviour. Setting this option false causes Exim to use only the host name. This should normally be done on a separate instance of the *smtp* transport, set up specially to handle the dialup hosts.

serialize_hosts	Use: *smtp*	Type: *host list*†	Default: *unset*

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 not usually 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. This can be done by setting **serialize_hosts** to match the relevant hosts.

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. The record is deleted when the connection is completed. Obviously there is scope for records to get 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.

If you set up this kind of serialization, you should also arrange to delete the relevant hints database whenever your system reboots. The names of the files start with *misc* and they are kept in the *spool/db* directory. There may be one or two files, depending on the type of DBM in use. The same files are used for ETRN serialization.

size_addition	Use: *smtp*	Type: *integer*	Default: *1024*

If a remote SMTP server indicates that it supports the SIZE option of the MAIL command, Exim uses this to pass over the message size at the start of an SMTP transaction. It adds the value of **size_addition** to the value it sends, 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.

Alternatively, if the value of **size_addition** is set negative, it disables the use of the SIZE option altogether.

tls_certificate	Use: *smtp*	Type: *string*†	Default: *unset*

The value of this option must be the absolute path to a file which contains the client's certificate, for possible use when sending a message over an encrypted connection. The values of *$host* and *$host_address* are set to the name and address of the server during the expansion. See chapter 39 for details of TLS.

Note: This option must be set if you want Exim to be able to use a TLS certificate when sending messages as a client. The global option of the same name specifies the certificate for Exim as a server; it is not automatically assumed that the same certificate should be used when Exim is operating as a client.

tls_crl	Use: *smtp*	Type: *string*†	Default: *unset*

This option specifies a certificate revocation list. The expanded value must be the name of a file that contains a CRL in PEM format.

tls_privatekey	Use: *smtp*	Type: *string†*	Default: *unset*

The value of this option must be the absolute path to a file which contains the client's private key. This is used when sending a message over an encrypted connection using a client certificate. The values of *$host* and *$host_address* are set to the name and address of the server during the expansion. If this option is unset, or the expansion is forced to fail, or the result is an empty string, the private key is assumed to be in the same file as the certificate. See chapter 39 for details of TLS.

tls_require_ciphers	Use: *smtp*	Type: *string†*	Default: *unset*

The value of this option must be a list of permitted cipher suites, for use when setting up an outgoing encrypted connection. (There is a global option of the same name for controlling incoming connections.) The values of *$host* and *$host_address* are set to the name and address of the server during the expansion. See chapter 39 for details of TLS; note that this option is used in different ways by OpenSSL and GnuTLS (see sections 39.4 and 39.5). For GnuTLS, the order of the ciphers is a preference order.

tls_tempfail_tryclear	Use: *smtp*	Type: *boolean*	Default: *true*

When the server host is not in **hosts_require_tls**, and there is a problem in setting up a TLS session, this option determines whether or not Exim should try to deliver the message unencrypted. If it is set false, delivery to the current host is deferred; if there are other hosts, they are tried. If this option is set true, Exim attempts to deliver unencrypted after a 4*xx* response to STARTTLS. Also, if STARTTLS is accepted, but the subsequent TLS negotiation fails, Exim closes the current connection (because it is in an unknown state), opens a new one to the same host, and then tries the delivery in clear.

tls_verify_certificates	Use: *smtp*	Type: *string†*	Default: *unset*

The value of this option must be the absolute path to a file containing permitted server certificates, for use when setting up an encrypted connection. Alternatively, if you are using OpenSSL, you can set **tls_verify_certificates** to the name of a directory containing certificate files. This does not work with GnuTLS; the option must be set to the name of a single file if you are using GnuTLS. The values of *$host* and *$host_address* are set to the name and address of the server during the expansion of this option. See chapter 39 for details of TLS.

30.5 How the limits for the number of hosts to try are used

There are two options that are concerned with the number of hosts that are tried when an SMTP delivery takes place. They are **hosts_max_try** and **hosts_max_try_hardlimit**.

The **hosts_max_try** option limits the number of hosts that are tried for a single delivery. However, despite the term "host" in its name, the option actually applies to each IP address independently. In other words, a multihomed host is treated as several independent hosts, just as it is for retrying.

Many of the larger ISPs have multiple MX records which often point to multihomed hosts. As a result, a list of a dozen or more IP addresses may be created as a result of routing one of these domains.

Trying every single IP address on such a long list does not seem sensible; if several at the top of the list fail, it is reasonable to assume there is some problem that is likely to affect all of them. Roughly speaking, the value of **hosts_max_try** is the maximum number that are tried before deferring the delivery. However, the logic cannot be quite that simple.

Firstly, IP addresses that are skipped because their retry times have not arrived do not count, and in addition, addresses that are past their retry limits are also not counted, even when they are tried. This means that when some IP addresses are past their retry limits, more than the value of **hosts_max_retry** may be tried. The reason for this behaviour is to ensure that all IP addresses are considered before timing out an email address (but see below for an exception).

Secondly, when the **hosts_max_try** limit is reached, Exim looks down the host list to see if there is a subsequent host with a different (higher valued) MX. If there is, that host is considered next, and the current IP address is used but not counted. This behaviour helps in the case of a domain with a retry rule that hardly ever delays any hosts, as is now explained:

Consider the case of a long list of hosts with one MX value, and a few with a higher MX value. If **hosts_max_try** is small (the default is 5) only a few hosts at the top of the list are tried at first. With the default retry rule, which specifies increasing retry times, the higher MX hosts are eventually tried when those at the top of the list are skipped because they have not reached their retry times.

However, it is common practice to put a fixed short retry time on domains for large ISPs, on the grounds that their servers are rarely down for very long. Unfortunately, these are exactly the domains that tend to resolve to long lists of hosts. The short retry time means that the lowest MX hosts are tried every time. The attempts may be in a different order because of random sorting, but without the special MX check, the higher MX hosts would never be tried until all the lower MX hosts had timed out (which might be several days), because there are always some lower MX hosts that have reached their retry times. With the special check, Exim considers at least one IP address from each MX value at every delivery attempt, even if the **hosts_max_try** limit has already been reached.

The above logic means that **hosts_max_try** is not a hard limit, and in particular, Exim normally eventually tries all the IP addresses before timing out an email address. When **hosts_max_try** was implemented, this seemed a reasonable thing to do. Recently, however, some lunatic DNS configurations have been set up with hundreds of IP addresses for some domains. It can take a very long time indeed for an address to time out in these cases.

The **hosts_max_try_hardlimit** option was added to help with this problem. Exim never tries more than this number of IP addresses; if it hits this limit and they are all timed out, the email address is bounced, even though not all possible IP addresses have been tried.

31. Address rewriting

There are some circumstances in which Exim automatically rewrites domains in addresses. The two most common are when an address is given without a domain (referred to as an "unqualified address") or when an address contains an abbreviated domain that is expanded by DNS lookup.

Unqualified envelope addresses are accepted only for locally submitted messages, or for messages that are received from hosts matching **sender_unqualified_hosts** or **recipient_unqualified_hosts**, as appropriate. Unqualified addresses in header lines are qualified if they are in locally submitted messages, or messages from hosts that are permitted to send unqualified envelope addresses. Otherwise, unqualified addresses in header lines are neither qualified nor rewritten.

One situation in which Exim does *not* automatically rewrite a domain is when it is the name of a CNAME record in the DNS. The older RFCs suggest that such a domain should be rewritten using the "canonical" name, and some MTAs do this. The new RFCs do not contain this suggestion.

31.1 Explicitly configured address rewriting

This chapter describes the rewriting rules that can be used in the main rewrite section of the configuration file, and also in the generic **headers_rewrite** option that can be set on any transport.

Some people believe that configured address rewriting is a Mortal Sin. Others believe that life is not possible without it. Exim provides the facility; you do not have to use it.

The main rewriting rules that appear in the "rewrite" section of the configuration file are applied to addresses in incoming messages, both envelope addresses and addresses in header lines. Each rule specifies the types of address to which it applies.

Whether or not addresses in header lines are rewritten depends on the origin of the headers and the type of rewriting. Global rewriting, that is, rewriting rules from the rewrite section of the configuration file, is applied only to those headers that were received with the message. Header lines that are added by ACLs or by a system filter or by individual routers or transports (which are specific to individual recipient addresses) are not rewritten by the global rules.

Rewriting at transport time, by means of the **headers_rewrite** option, applies all headers except those added by routers and transports. That is, as well as the headers that were received with the message, it also applies to headers that were added by an ACL or a system filter.

In general, rewriting addresses from your own system or domain 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 it can sometimes be used as a routing tool, this is very strongly discouraged.

There are two commonly encountered circumstances where rewriting is used, as illustrated by these examples:

- The company whose domain is *hitch.fict.example* has a number of hosts that exchange mail with each other behind a firewall, but there is only a single gateway to the outer world. The gateway rewrites **.hitch.fict.example* as *hitch.fict.example* when sending mail off-site.
- A host rewrites the local parts of its own users so that, for example, *fp42@hitch.fict.example* becomes *Ford.Prefect@hitch.fict.example*.

31.2 When does rewriting happen?

Configured address rewriting can take place at several different stages of a message's processing.

At the start of an ACL for MAIL, the sender address may have been rewritten by a special SMTP-time rewrite rule (see section 31.9), but no ordinary rewrite rules have yet been applied. If, however, the sender address is verified in the ACL, it is rewritten before verification, and remains rewritten thereafter. The subsequent value of *$sender_address* is the rewritten address. This also applies if sender verification happens in a RCPT ACL. Otherwise, when the sender address is not verified, it is rewritten as soon as a message's header lines have been received.

Similarly, at the start of an ACL for RCPT, the current recipient's address may have been rewritten by a special SMTP-time rewrite rule, but no ordinary rewrite rules have yet been applied to it. However, the behaviour is different from the sender address when a recipient is verified. The address is rewritten for the verification, but the rewriting is not remembered at this stage. The value of *$local_part* and *$domain* after verification are always the same as they were before (that is, they contain the unrewritten – except for SMTP-time rewriting – address).

As soon as a message's header lines have been received, all the envelope recipient addresses are permanently rewritten, and rewriting is also applied to the addresses in the header lines (if configured). This happens before adding any header lines that were specified in MAIL or RCPT ACLs, and before the DATA ACL and *local_scan()* functions are run.

When an address is being routed, either for delivery or for verification, rewriting is applied immediately to child addresses that are generated by redirection, unless **no_rewrite** is set on the router.

At transport time, additional rewriting of addresses in header lines can be specified by setting the generic **headers_rewrite** option on a transport. This option contains rules that are identical in form to those in the rewrite section of the configuration file. They are applied to the original message header lines and any that were added by ACLs or a system filter. They are not applied to header lines that are added by routers or the transport.

The outgoing envelope sender can be rewritten by means of the **return_path** transport option. However, it is not possible to rewrite envelope recipients at transport time.

31.3 Testing the rewriting rules that apply on input

Exim's input rewriting configuration appears in a part of the run time configuration file headed by "begin rewrite". It can be tested by the **-brw** command line option. This takes an address (which can be a full RFC 2822 address) as its argument. The output is a list of how the address would be transformed by the rewriting rules for each of the different places it might appear in an incoming message, that is, for each different header and for the envelope sender and recipient fields. For example,

```
exim -brw ph10@exim.workshop.example
```

might produce the output

```
sender: Philip.Hazel@exim.workshop.example
from: Philip.Hazel@exim.workshop.example
to: ph10@exim.workshop.example
cc: ph10@exim.workshop.example
bcc: ph10@exim.workshop.example
reply-to: Philip.Hazel@exim.workshop.example
env-from: Philip.Hazel@exim.workshop.example
env-to: ph10@exim.workshop.example
```

which shows that 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. At the present time, there is no equivalent way of testing rewriting rules that are set for a particular transport.

31.4 Rewriting rules

The rewrite section of the configuration file consists of lines of rewriting rules in the form

<source pattern> <replacement> <flags>

Rewriting rules that are specified for the **headers_rewrite** generic transport option are given as a colon-separated list. Each item in the list takes the same form as a line in the main rewriting configuration (except that any colons must be doubled, of course).

The formats of source patterns and replacement strings are described below. Each is terminated by white space, unless enclosed in double quotes, in which case normal quoting conventions apply inside

the quotes. The flags are single characters which may appear in any order. Spaces and tabs between them are ignored.

For each address that could potentially be rewritten, the rules are scanned in order, and replacements for the address from earlier rules can themselves be replaced by later rules (but see the "q" and "R" flags).

The order in which addresses are rewritten is undefined, may change between releases, and must not be relied on, with one exception: when a message is received, the envelope sender is always rewritten first, before any header lines are rewritten. For example, the replacement string for a rewrite of an address in *To:* must not assume that the message's address in *From:* has (or has not) already been rewritten. However, a rewrite of *From:* may assume that the envelope sender has already been rewritten.

The variables *$local_part* and *$domain* can be used in the replacement string to refer to the address that is being rewritten. Note that lookup-driven rewriting can be done by a rule of the form

```
*@*   ${lookup ...
```

where the lookup key uses *$1* and *$2* or *$local_part* and *$domain* to refer to the address that is being rewritten.

31.5 Rewriting patterns

The source pattern in a rewriting rule is any item which may appear in an address list (see section 10.19). It is in fact processed as a single-item address list, which means that it is expanded before being tested against the address. As always, if you use a regular expression as a pattern, you must take care to escape dollar and backslash characters, or use the `\N` facility to suppress string expansion within the regular expression.

Domains in patterns should be given in lower case. Local parts in patterns are case-sensitive. If you want to do case-insensitive matching of local parts, you can use a regular expression that starts with `^(?i)`.

After matching, the numerical variables *$1*, *$2*, etc. may be set, depending on the type of match which occurred. These can be used in the replacement string to insert portions of the incoming address. *$0* always refers to the complete incoming address. When a regular expression is used, the numerical variables are set from its capturing subexpressions. For other types of pattern they are set as follows:

- If a local part or domain starts with an asterisk, the numerical variables refer to the character strings matched by asterisks, with *$1* associated with the first asterisk, and *$2* with the second, if present. For example, if the pattern

  ```
  *queen@*.fict.example
  ```

  is matched against the address *hearts-queen@wonderland.fict.example* then

  ```
  $0 = hearts-queen@wonderland.fict.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.

- If the domain part of the pattern is a partial lookup, the wild and fixed parts of the domain are placed in the next available numerical variables. Suppose, for example, that the address *foo@bar.baz.example* is processed by a rewriting rule of the form

  ```
  *@partial-dbm;/some/dbm/file  <replacement string>
  ```

  and the key in the file that matches the domain is `*.baz.example`. Then

  ```
  $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 non-wild key is matched in a partial lookup, *$2* is again set to the empty string and *$3* is set to the whole domain. For non-partial domain lookups, no numerical variables are set.

31.6 Rewriting replacements

If the replacement string for a rule is a single asterisk, addresses that match the pattern and the flags are *not* rewritten, and no subsequent rewriting rules are scanned. For example,

```
hatta@lookingglass.fict.example  *  f
```

specifies that *hatta@lookingglass.fict.example* is never to be rewritten in *From:* headers.

If the replacement string is not a single asterisk, it is expanded, and must yield a fully qualified address. Within the expansion, the variables *$local_part* and *$domain* refer to the address that is being rewritten. Any letters they contain retain their original case – they are not lower cased. The numerical variables are set up according to the type of pattern that matched the address, as described above. If the expansion is forced to fail by the presence of "fail" in a conditional or lookup item, rewriting by the current rule is abandoned, but subsequent rules may take effect. Any other expansion failure causes the entire rewriting operation to be abandoned, and an entry written to the panic log.

31.7 Rewriting flags

There are three different kinds of flag that may appear on rewriting rules:

- Flags that specify which headers and envelope addresses to rewrite: E, F, T, b, c, f, h, r, s, t.
- A flag that specifies rewriting at SMTP time: S.
- Flags that control the rewriting process: Q, q, R, w.

For rules that are part of the **headers_rewrite** generic transport option, E, F, T, and S are not permitted.

31.8 Flags specifying which headers and envelope addresses to rewrite

If none of the following flag letters, nor the "S" flag (see section 31.9) are present, a main rewriting rule applies to all headers and to both the sender and recipient fields of the envelope, whereas a transport-time rewriting rule just applies to all headers. Otherwise, the rewriting rule is skipped unless the relevant addresses are being processed.

`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
`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

You should be particularly careful about rewriting *Sender:* headers, and restrict this to special known cases in your own domains.

31.9 The SMTP-time rewriting flag

The rewrite flag "S" specifies a rewrite of incoming envelope addresses at SMTP time, as soon as an address is received in a MAIL or RCPT command, and before any other processing; even before syntax checking. The pattern is required to be a regular expression, and it is matched against the whole of the data for the command, including any surrounding angle brackets.

This form of rewrite rule allows for the handling of addresses that are not compliant with RFCs 2821 and 2822 (for example, "bang paths" in batched SMTP input). Because the input is not required to be a syntactically valid address, the variables *$local_part* and *$domain* are not available during the expansion of the replacement string. The result of rewriting replaces the original address in the MAIL or RCPT command.

31.10 Flags controlling the rewriting process

There are four flags which 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 flags) and 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.
- If the "q" flag is set on a rule, no further rewriting rules are considered, even if no rewriting actually takes place because of a "fail" in the expansion. The "q" flag is not effective if the address is of the wrong type (does not match the flags) or does not match the pattern.
- The "R" flag causes a successful rewriting rule to be re-applied to the new address, up to ten times. It can be combined with the "q" flag, to stop rewriting once it fails to match (after at least one successful rewrite).
- When an address in a header is rewritten, the rewriting normally applies only to the working part of the address, with any comments and RFC 2822 "phrase" left unchanged. For example, rewriting might change

  ```
  From: Ford Prefect <fp42@restaurant.hitch.fict.example>
  ```

  into

  ```
  From: Ford Prefect <prefectf@hitch.fict.example>
  ```

  Sometimes there is a need to replace the whole 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 entire address is replaced, not just the working part. The replacement must be a complete RFC 2822 address, including the angle brackets if necessary. If text outside angle brackets contains a character whose value is greater than 126 or less than 32 (except for tab), the text is encoded according to RFC 2047. The character set is taken from **headers_charset**, which defaults to ISO-8859-1.

  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.

31.11 Rewriting examples

Here is an example of the two common rewriting paradigms:

```
*@*.hitch.fict.example  $1@hitch.fict.example
*@hitch.fict.example    ${lookup{$1}dbm{/etc/realnames}\
                      {$value}fail}@hitch.fict.example bctfrF
```

Note the use of "fail" in the lookup expansion in the second rule, forcing the string expansion to fail if the lookup does not succeed. In this context it has the effect of leaving the original address unchanged, but Exim goes on to consider subsequent rewriting rules, if any, because the "q" flag is not present in that rule. An alternative to "fail" would be to supply *$1* explicitly, which would cause the rewritten address to be the same as before, at the cost of a small bit of processing. Not supplying either of these is an error, since the rewritten address would then contain no local part.

The first example above replaces the domain with a superior, more general domain. This may not be desirable for certain local parts. If the rule

```
root@*.hitch.fict.example  *
```

were inserted before the first rule, rewriting would be suppressed for the local part *root* at any domain ending in *hitch.fict.example*.

Rewriting can be made conditional on a number of tests, by making use of *${if* in the expansion item. For example, to apply a rewriting rule only to messages that originate outside the local host:

```
*@*.hitch.fict.example  "${if !eq {$sender_host_address}{}\
                         {$1@hitch.fict.example}fail}"
```

The replacement string is quoted in this example because it contains white space.

Exim does not handle addresses in the form of "bang paths". If it sees such an address it treats it as an unqualified local part which it qualifies with the local qualification domain (if the source of the message is local or if the remote host is permitted to send unqualified addresses). Rewriting can sometimes be used to handle simple bang paths with a fixed number of components. For example, the rule

```
\N^([^!]+)!(.*)@your.domain.example$\N   $2@$1
```

rewrites a two-component bang path *host.name!user* as the domain address *user@host.name*. However, there is a security implication in using this as a global rewriting rule for envelope addresses. It can provide a backdoor method for using your system as a relay, because the incoming addresses appear to be local. If the bang path addresses are received via SMTP, it is safer to use the "S" flag to rewrite them as they are received, so that relay checking can be done on the rewritten addresses.

32. Retry configuration

The "retry" section of the runtime configuration file contains a list of retry rules that control how often Exim tries to deliver messages that cannot be delivered at the first attempt. If there are no retry rules (the section is empty or not present), there are no retries. In this situation, temporary errors are treated as permanent. The default configuration contains a single, general-purpose retry rule (see section 7.5). The **-brt** command line option can be used to test which retry rule will be used for a given address, domain and error.

The most common cause of retries is temporary failure to deliver to a remote host because the host is down, or inaccessible because of a network problem. Exim's retry processing in this case is applied on a per-host (strictly, per IP address) basis, not on a per-message basis. Thus, if one message has recently been delayed, delivery of a new message to the same host is not immediately tried, but waits for the host's retry time to arrive. If the **retry_defer** log selector is set, the message "retry time not reached" is written to the main log whenever a delivery is skipped for this reason. Section 45.2 contains more details of the handling of errors during remote deliveries.

Retry processing applies to routing as well as to delivering, except as covered in the next paragraph. The retry rules do not distinguish between these actions. It is not possible, for example, to specify different behaviour for failures to route the domain *snark.fict.example* and failures to deliver to the host *snark.fict.example*. I didn't think anyone would ever need this added complication, so did not implement it. However, although they share the same retry rule, the actual retry times for routing and transporting a given domain are maintained independently.

When a delivery is not part of a queue run (typically an immediate delivery on receipt of a message), the routers are always run, and local deliveries are always attempted, even if retry times are set for them. This makes for better behaviour 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.

32.1 Changing retry rules

If you change the retry rules in your configuration, you should consider whether or not to delete the retry data that is stored in Exim's spool area in files with names like *db/retry*. Deleting any of Exim's hints files is always safe; that is why they are called "hints".

The hints retry data contains suggested retry times based on the previous rules. In the case of a long-running problem with a remote host, it might record the fact that the host has timed out. If your new rules increase the timeout time for such a host, you should definitely remove the old retry data and let Exim recreate it, based on the new rules. Otherwise Exim might bounce messages that it should now be retaining.

32.2 Format of retry rules

Each retry rule occupies one line and consists of three or four parts, separated by white space: a pattern, an error name, an optional list of sender addresses, and a list of retry parameters. The pattern and sender lists must be enclosed in double quotes if they contain white space. The rules are searched in order until one is found where the pattern, error name, and sender list (if present) match the failing host or address, the error that occurred, and the message's sender, respectively.

The pattern is any single item that may appear in an address list (see section 10.19). It is in fact processed as a one-item address list, which means that it is expanded before being tested against the address that has been delayed. A negated address list item is permitted. Address list processing treats a plain domain name as if it were preceded by "*@", which makes it possible for many retry rules to start with just a domain. For example,

```
lookingglass.fict.example        *  F,24h,30m;
```

provides a rule for any address in the *lookingglass.fict.example* domain, whereas

```
alice@lookingglass.fict.example  *  F,24h,30m;
```

applies only to temporary failures involving the local part **alice**. In practice, almost all rules start with a domain name pattern without a local part.

Warning: If you use a regular expression in a routing rule pattern, it must match a complete address, not just a domain, because that is how regular expressions work in address lists.

```
^\Nxyz\d+\.abc\.example$\N         *  G,1h,10m,2   Wrong
^\N[^@]+@xyz\d+\.abc\.example$\N   *  G,1h,10m,2   Right
```

32.3 Choosing which retry rule to use for address errors

When Exim is 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 against the complete address only if **retry_use_local_part** is set for the router. Otherwise, only the domain is used, except when matching against a regular expression, when the local part of the address is replaced with "*". A domain on its own can match a domain pattern, or a pattern that starts with "*@". By default, **retry_use_local_part** is true for routers where **check_local_user** is true, and false for other routers.

Similarly, when Exim is looking for a retry rule after a local delivery has failed (for example, after a mailbox full error), each line in the retry configuration is tested against the complete address only if **retry_use_local_part** is set for the transport (it defaults true for all local transports).

However, when Exim is looking for a retry rule after a remote delivery attempt suffers an address error (a 4*xx* SMTP response for a recipient address), the whole address is always used as the key when searching the retry rules. The rule that is found is used to create a retry time for the combination of the failing address and the message's sender. It is the combination of sender and recipient that is delayed in subsequent queue runs until its retry time is reached. You can delay the recipient without regard to the sender by setting **address_retry_include_sender** false in the *smtp* transport but this can lead to problems with servers that regularly issue 4*xx* responses to RCPT commands.

32.4 Choosing which retry rule to use for host and message errors

For a temporary error that is not related to an individual address (for example, a connection timeout), each line in the retry configuration is checked twice. First, the name of the remote host is used as a domain name (preceded by "*@" when matching a regular expression). If this does not match the line, the domain from the email address is tried in a similar fashion. For example, suppose the MX records for *a.b.c.example* are

```
a.b.c.example  MX  5  x.y.z.example
               MX  6  p.q.r.example
               MX  7  m.n.o.example
```

and the retry rules are

```
p.q.r.example    *      F,24h,30m;
a.b.c.example    *      F,4d,45m;
```

and a delivery to the host *x.y.z.example* suffers a connection failure. The first rule matches neither the host nor the domain, so Exim looks at the second rule. This does not match the host, but it does match the domain, so it is used to calculate the retry time for the host *x.y.z.example*. Meanwhile, Exim tries to deliver to *p.q.r.example*. If this also suffers a host error, the first retry rule is used, because it matches the host.

In other words, temporary failures to deliver to host *p.q.r.example* use the first rule to determine retry times, but for all the other hosts for the domain *a.b.c.example*, the second rule is used. The second rule is also used if routing to *a.b.c.example* suffers a temporary failure.

Note: The host name is used when matching the patterns, not its IP address. However, if a message is routed directly to an IP address without the use of a host name, for example, if a *manualroute* router contains a setting such as:

```
route_list = *.a.example  192.168.34.23
```

then the “host name” that is used when searching for a retry rule is the textual form of the IP address.

32.5 Retry rules for specific errors

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 tested for are:

auth_failed
Authentication failed when trying to send to a host in the **hosts_require_auth** list in an *smtp* transport.

data_4xx
A 4*xx* error was received for an outgoing DATA command, either immediately after the command, or after sending the message's data.

mail_4xx
A 4*xx* error was received for an outgoing MAIL command.

rcpt_4xx
A 4*xx* error was received for an outgoing RCPT command.

For the three 4*xx* errors, either the first or both of the x's can be given as specific digits, for example: `mail_45x` or `rcpt_436`. For example, to recognize 452 errors given to RCPT commands for addresses in a certain domain, and have retries every ten minutes with a one-hour timeout, you could set up a retry rule of this form:

```
the.domain.name  rcpt_452   F,1h,10m
```

These errors apply to both outgoing SMTP (the *smtp* transport) and outgoing LMTP (either the *lmtp* transport, or the *smtp* transport in LMTP mode).

lost_connection
A server unexpectedly closed the SMTP connection. There may, of course, legitimate reasons for this (host died, network died), but if it repeats a lot for the same host, it indicates something odd.

refused_MX
A connection to a host obtained from an MX record was refused.

refused_A
A connection to a host not obtained from an MX record was refused.

refused
A connection was refused.

timeout_connect_MX
A connection attempt to a host obtained from an MX record timed out.

timeout_connect_A
A connection attempt to a host not obtained from an MX record timed out.

timeout_connect
A connection attempt timed out.

timeout_MX
There was a timeout while connecting or during an SMTP session with a host obtained from an MX record.

timeout_A
There was a timeout while connecting or during an SMTP session with a host not obtained from an MX record.

timeout
There was a timeout while connecting or during an SMTP session.

tls_required
The server was required to use TLS (it matched **hosts_require_tls** in the *smtp* transport), but either did not offer TLS, or it responded with 4*xx* to STARTTLS, or there was a problem setting up the TLS connection.

quota
A mailbox quota was exceeded in a local delivery by the *appendfile* transport.

quota_<*time*>
A mailbox quota was exceeded in a local delivery by the *appendfile* transport, and the mailbox has not been accessed for <*time*>. For example, *quota_4d* applies to a quota error when the mailbox has not been accessed for four days.

The idea of **quota_**<*time*> is to make it possible to have shorter timeouts when the mailbox is full and is not being read by its owner. Ideally, it should be based on the last time that the user accessed the mailbox. However, it is not always possible to determine this. Exim uses the following heuristic rules:

- If the mailbox is a single file, the time of last access (the "atime") is used. As no new messages are being delivered (because the mailbox is over quota), Exim does not access the file, so this is the time of last user access.
- For a maildir delivery, the time of last modification of the *new* subdirectory is used. As the mailbox is over quota, no new files are created in the *new* subdirectory, because no new messages are being delivered. Any change to the *new* subdirectory is therefore assumed to be the result of an MUA moving a new message to the *cur* directory when it is first read. The time that is used is therefore the last time that the user read a new message.
- For other kinds of multi-file mailbox, the time of last access cannot be obtained, so a retry rule that uses this type of error field is never matched.

The quota errors apply both to system-enforced quotas and to Exim's own quota mechanism in the *appendfile* transport. The *quota* error also applies when a local delivery is deferred because a partition is full (the ENOSPC error).

32.6 Retry rules for specified senders

You can specify retry rules that apply only when the failing message has a specific sender. In particular, this can be used to define retry rules that apply only to bounce messages. The third item in a retry rule can be of this form:

```
senders=<address list>
```

The retry timings themselves are then the fourth item. For example:

```
*   rcpt_4xx   senders=:   F,1h,30m
```

matches recipient 4*xx* errors for bounce messages sent to any address at any host. If the address list contains white space, it must be enclosed in quotes. For example:

```
a.domain  rcpt_452  senders="xb.dom : yc.dom"  G,8h,10m,1.5
```

Warning: This facility can be unhelpful if it is used for host errors (which do not depend on the recipient). The reason is that the sender is used only to match the retry rule. Once the rule has been found for a host error, its contents are used to set a retry time for the host, and this will apply to all messages, not just those with specific senders.

When testing retry rules using **-brt**, you can supply a sender using the **-f** command line option, like this:

```
exim -f "" -brt user@dom.ain
```

If you do not set **-f** with **-brt**, a retry rule that contains a senders list is never matched.

32.7 Retry parameters

The third (or fourth, if a senders list is present) field in a retry rule is a sequence of retry parameter sets, separated by semicolons. Each set consists of

<*letter*>,<*cutoff time*>,<*arguments*>

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. The cutoff time is measured from the time that the first failure for the domain (combined with the local part if relevant) was detected, not from the time the message was received.

The available algorithms are:

- *F*: retry at fixed intervals. There is a single time parameter specifying the interval.
- *G*: retry at geometrically increasing intervals. The first argument specifies a starting value for the interval, and the second a multiplier, which is used to increase the size of the interval at each retry.
- *H*: retry at randomized intervals. The arguments are as for *G*. For each retry, the previous interval is multiplied by the factor in order to get a maximum for the next interval. The minimum interval is the first argument of the parameter, and an actual interval is chosen randomly between them. Such a rule has been found to be helpful in cluster configurations when all the members of the cluster restart at once, and may therefore synchronize their queue processing times.

When computing the next retry time, the algorithm definitions are scanned in order until one whose cutoff time has not yet passed is reached. This is then used to compute a new retry time that is later than the current time. In the case of fixed interval retries, this simply means adding the 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. The main configuration variable **retry_interval_max** limits the maximum interval between retries. It cannot be set greater than `24h`, which is its default value.

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 unusable, Exim will generate retry times for it 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.

Retry times 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 attempt new deliveries only for those addresses that have passed their next retry time. If a new message arrives for a deferred address, an immediate delivery attempt occurs only if the address has passed its retry time. In the absence of new messages, the minimum time between retries is the interval between queue runner processes. There is not much point in setting retry times of five 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).

The data in the retry hints database can be inspected by using the *exim_dumpdb* or *exim_fixdb* utility programs (see chapter 50). The latter utility can also be used to change the data. The *exinext* utility script can be used to find out what the next retry times are for the hosts associated with a particular mail domain, and also for local deliveries that have been deferred.

32.8 Retry rule examples

Here are some example retry rules:

```
alice@wonderland.fict.example quota_5d  F,7d,3h
wonderland.fict.example       quota_5d
wonderland.fict.example       *         F,1h,15m; G,2d,1h,2;
lookingglass.fict.example     *         F,24h,30m;
*                 refused_A   F,2h,20m;
*                 *           F,2h,15m; G,16h,1h,1.5; F,5d,8h
```

The first rule sets up special handling for mail to *alice@wonderland.fict.example* when there is an over-quota error and the mailbox has not been read for at least 5 days. Retries continue every three hours for 7 days. The second rule handles over-quota errors for all other local parts at *wonderland.fict.example*; the absence of a local part has the same effect as supplying "*@". As no retry algorithms are supplied, messages that fail are bounced immediately if the mailbox has not been read for at least 5 days.

The third rule handles all other errors at *wonderland.fict.example*; retries happen every 15 minutes for an hour, then with geometrically increasing intervals until two days have passed since a delivery first failed. After the first hour there is a delay of one hour, then two hours, then four hours, and so on (this is a rather extreme example).

The fourth rule controls retries for the domain *lookingglass.fict.example*. They happen every 30 minutes for 24 hours only. 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 final rule in a retry configuration should always have asterisks in the first two fields so as to provide a general catch-all for any addresses that do not have their own special handling. This example tries every 15 minutes for 2 hours, then with intervals starting at one hour and increasing by a factor of 1.5 up to 16 hours, then every 8 hours up to 5 days.

32.9 Timeout of retry data

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 to deliver to it immediately a message arrives, and if that fails, it will calculate a retry time as if it were failing for the first time.

This improves the behaviour for messages routed to rarely-used hosts such as MX backups. If such a host was down at one time, and happens to be down again when Exim tries a month later, using the old retry data would imply that it had been down all the time, which is not a justified assumption.

If a host really is permanently dead, this behaviour 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.

32.10 Long-term failures

Special processing happens when an email address has been failing for so long that the cutoff time for the last algorithm is reached. For example, using the default retry rule:

```
* * F,2h,15m; G,16h,1h,1.5; F,4d,6h
```

the cutoff time is four days. Reaching the retry cutoff is independent of how long any specific message has been failing; it is the length of continuous failure for the recipient address that counts.

When the cutoff time is reached for a local delivery, or for all the IP addresses associated with a remote delivery, a subsequent delivery failure causes Exim to give up on the address, and a bounce message is generated. In order to cater for new messages that use the failing address, a next retry time is still computed from the final algorithm, and is used as follows:

For local deliveries, one delivery attempt is always made for any subsequent messages. If this delivery fails, the address fails immediately. The post-cutoff retry time is not used.

If the delivery is remote, there are two possibilities, controlled by the **delay_after_cutoff** option of the *smtp* transport. The option is true by default. Until the post-cutoff retry time for one of the IP addresses is reached, the failing email address is bounced immediately, without a delivery attempt taking place. After that time, 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.

In other words, when all the hosts for a given email address have been failing for a long time, Exim bounces rather then defers until one of the hosts' retry times is reached. Then it tries once, and

bounces if that attempt fails. This behaviour ensures that few resources are wasted in repeatedly trying to deliver to a broken destination, but if the host does recover, Exim will eventually notice.

If **delay_after_cutoff** is set false, Exim behaves differently. If all IP addresses 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 no suitable IP addresses, 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 addresses immediately, unless they have been tried since the message arrived. If there is a continuous stream of messages for the failing domains, setting **delay_after_cutoff** false means that there will be many more attempts to deliver to permanently failing IP addresses than when **delay_after_cutoff** is true.

32.11 Deliveries that work intermittently

Some additional logic is needed to cope with cases 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. In this situation, because some messages are successfully delivered, the "retry clock" for the host or address keeps getting reset by the successful deliveries, and so failing messages remain on the queue for ever because the cutoff time is never reached.

Two exceptional actions are applied to prevent this happening. The first applies to errors that are related to a message rather than a remote host. Section 45.2 has a discussion of the different kinds of error; examples of message-related errors are 4*xx* responses to MAIL or DATA commands, and quota failures. For this type of error, if a message's arrival time is earlier than the "first failed" time for the error, the earlier time is used when scanning the retry rules to decide when to try next and when to time out the address.

The exceptional second action applies in all cases. If a message has been on the queue for longer than the cutoff time of any applicable retry rule for a given address, a delivery is attempted for that address, even if it is not yet time, and if this delivery fails, the address is timed out. A new retry time is not computed in this case, so that other messages for the same address are considered immediately.

33. SMTP authentication

The "authenticators" section of Exim's run time configuration is concerned with SMTP authentication. This facility is an extension to the SMTP protocol, described in RFC 2554, which allows a client SMTP host to authenticate itself to a server. This is a common way for a server to recognize clients that are permitted to use it as a relay. SMTP authentication is not of relevance to the transfer of mail between servers that have no managerial connection with each other.

Very briefly, the way SMTP authentication works is as follows:

- The server advertises a number of authentication *mechanisms* in response to the client's EHLO command.
- The client issues an AUTH command, naming a specific mechanism. The command may, optionally, contain some authentication data.
- The server may issue one or more *challenges*, to which the client must send appropriate responses. In simple authentication mechanisms, the challenges are just prompts for user names and passwords. The server does not have to issue any challenges – in some mechanisms the relevant data may all be transmitted with the AUTH command.
- The server either accepts or denies authentication.
- If authentication succeeds, the client may optionally make use of the AUTH option on the MAIL command to pass an authenticated sender in subsequent mail transactions. Authentication lasts for the remainder of the SMTP connection.
- If authentication fails, the client may give up, or it may try a different authentication mechanism, or it may try transferring mail over the unauthenticated connection.

If you are setting up a client, and want to know which authentication mechanisms the server supports, you can use Telnet to connect to port 25 (the SMTP port) on the server, and issue an EHLO command. The response to this includes the list of supported mechanisms. For example:

```
$ telnet server.example 25
Trying 192.168.34.25...
Connected to server.example.
Escape character is '^]'.
220 server.example ESMTP Exim 4.20 ...
ehlo client.example
250-server.example Hello client.example [10.8.4.5]
250-SIZE 52428800
250-PIPELINING
250-AUTH PLAIN
250 HELP
```

The second-last line of this example output shows that the server supports authentication using the PLAIN mechanism. In Exim, the different authentication mechanisms are configured by specifying *authenticator* drivers. Like the routers and transports, which authenticators are included in the binary is controlled by build-time definitions. The following are currently available, included by setting

```
AUTH_CRAM_MD5=yes
AUTH_CYRUS_SASL=yes
AUTH_PLAINTEXT=yes
AUTH_SPA=yes
```

in *Local/Makefile*, respectively. The first of these supports the CRAM-MD5 authentication mechanism (RFC 2195), and the second provides an interface to the Cyrus SASL authentication library. The third can be configured to support the PLAIN authentication mechanism (RFC 2595) or the LOGIN mechanism, which is not formally documented, but used by several MUAs. The fourth authenticator supports Microsoft's *Secure Password Authentication* mechanism.

The authenticators are configured using the same syntax as other drivers (see section 6.22). If no authenticators are required, no authentication section need be present in the configuration file. Each authenticator can in principle have both server and client functions. 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. Authenticator configuration options are provided for use in both these circumstances.

To make it clear which options apply to which situation, 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 = ${if eq{$auth1}{ph10}{secret1}fail}
  client_name = ph10
  client_secret = secret2
```

The **server_** option is used when Exim is acting as a server, and the **client_** options when it is acting as a client.

Descriptions of the individual authenticators are given in subsequent chapters. The remainder of this chapter covers the generic options for the authenticators, followed by general discussion of the way authentication works in Exim.

33.1 Generic options for authenticators

client_condition	Use: *authenticators*	Type: *string*†	Default: *unset*

When Exim is authenticating as a client, it skips any authenticator whose **client_condition** expansion yields "0", "no", or "false". This can be used, for example, to skip plain text authenticators when the connection is not encrypted by a setting such as:

```
client_condition = ${if !eq{$tls_cipher}{}}
```

(Older documentation incorrectly states that *$tls_cipher* contains the cipher used for incoming messages. In fact, during SMTP delivery, it contains the cipher used for the delivery.)

driver	Use: *authenticators*	Type: *string*	Default: *unset*

This option must always be set. It specifies which of the available authenticators is to be used.

public_name	Use: *authenticators*	Type: *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 upper case 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's instance name.

server_advertise_condition	Use: *authenticators*	Type: *string*†	Default: *unset*

When a server is about to advertise an authentication mechanism, the condition is expanded. If it yields the empty string, "0", "no", or "false", the mechanism is not advertised. If the expansion fails, the mechanism is not advertised. If the failure was not forced, and was not caused by a lookup defer, the incident is logged. See section 33.3 below for further discussion.

server_condition	Use: *authenticators*	Type: *string*†	Default: *unset*

This option must be set for a **plaintext** server authenticator, where it is used directly to control authentication. See section 34.2 for details.

For the other authenticators, **server_condition** can be used as an additional authentication or authorization mechanism that is applied after the other authenticator conditions succeed. If it is set, it is expanded when the authenticator would otherwise return a success code. If the expansion is forced to fail, authentication fails. Any other expansion failure 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. For any other result, a temporary error code is returned, with the expanded string as the error text.

server_debug_print	Use: *authenticators*	Type: *string*†	Default: *unset*

If this option is set and authentication debugging is enabled (see the **-d** command line option), the string is expanded and included in the debugging output when the authenticator is run as a server. This can help with checking out the values of variables. If expansion of the string fails, the error message is written to the debugging output, and Exim carries on processing.

server_set_id	Use: *authenticators*	Type: *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 user name that was used to authenticate, and refer to it subsequently during delivery of the message. If expansion fails, the option is ignored.

server_mail_auth_condition	Use: *authenticators*	Type: *string*†	Default: *unset*

This option allows a server to discard authenticated sender addresses supplied as part of MAIL commands in SMTP connections that are authenticated by the driver on which **server_mail_auth_condition** is set. The option is not used as part of the authentication process; instead its (unexpanded) value is remembered for later use. How it is used is described in the following section.

33.2 The AUTH parameter on MAIL commands

When a client supplied an AUTH= item on a MAIL command, Exim applies the following checks before accepting it as the authenticated sender of the message:

- If the connection is not using extended SMTP (that is, HELO was used rather than EHLO), the use of AUTH= is a syntax error.
- If the value of the AUTH= parameter is "<>", it is ignored.
- If **acl_smtp_mailauth** is defined, the ACL it specifies is run. While it is running, the value of *$authenticated_sender* is set to the value obtained from the AUTH= parameter. If the ACL does not yield "accept", the value of *$authenticated_sender* is deleted. The **acl_smtp_mailauth** ACL may not return "drop" or "discard". If it defers, a temporary error code (451) is given for the MAIL command.
- If **acl_smtp_mailauth** is not defined, the value of the AUTH= parameter is accepted and placed in *$authenticated_sender* only if the client has authenticated.
- If the AUTH= value was accepted by either of the two previous rules, and the client has authenticated, and the authenticator has a setting for the **server_mail_auth_condition**, the condition is checked at this point. The valued that was saved from the authenticator is expanded. If the expansion fails, or yields an empty string, "0", "no", or "false", the value of *$authenticated_sender*

is deleted. If the expansion yields any other value, the value of *$authenticated_sender* is retained and passed on with the message.

When *$authenticated_sender* is set for a message, it is passed on to other hosts to which Exim authenticates as a client. Do not confuse this value with *$authenticated_id*, which is a string obtained from the authentication process, and which is not usually a complete email address.

Whenever an AUTH= value is ignored, the incident is logged. The ACL for MAIL, if defined, is run after AUTH= is accepted or ignored. It can therefore make use of *$authenticated_sender*. The converse is not true: the value of *$sender_address* is not yet set up when the **acl_smtp_mailauth** ACL is run.

33.3 Authentication on an Exim server

When Exim receives an EHLO command, it advertises the public names of those authenticators that are configured as servers, subject to the following conditions:

- The client host must match **auth_advertise_hosts** (default *).
- It the **server_advertise_condition** option is set, its expansion must not yield the empty string, “0”, “no”, or “false”.

The order in which the authenticators are defined controls the order in which the mechanisms are advertised.

Some mail clients (for example, some versions of Netscape) require the user to provide a name and password for authentication whenever AUTH is advertised, even though authentication may not in fact be needed (for example, Exim may be set up to allow unconditional relaying from the client by an IP address check). You can make such clients more friendly by not advertising AUTH to them. For example, if clients on the 10.9.8.0/24 network are permitted (by the ACL that runs for RCPT) to relay without authentication, you should set

```
auth_advertise_hosts = ! 10.9.8.0/24
```

so that no authentication mechanisms are advertised to them.

The **server_advertise_condition** controls the advertisement of individual authentication mechanisms. For example, it can be used to restrict the advertisement of a particular mechanism to encrypted connections, by a setting such as:

```
server_advertise_condition = ${if eq{$tls_cipher}{}{no}{yes}}
```

If the session is encrypted, *$tls_cipher* is not empty, and so the expansion yields “yes”, which allows the advertisement to happen.

When an Exim server receives an AUTH command from a client, it rejects it immediately if AUTH was not advertised in response to an earlier EHLO command. This is the case if

- The client host does not match **auth_advertise_hosts**; or
- No authenticators are configured with server options; or
- Expansion of **server_advertise_condition** blocked the advertising of all the server authenticators.

Otherwise, Exim runs the ACL specified by **acl_smtp_auth** in order to decide whether to accept the command. If **acl_smtp_auth** is not set, AUTH is accepted from any client host.

If AUTH is not rejected by the ACL, Exim searches its configuration for a server authentication mechanism that was advertised in response to EHLO and that matches the one named in the AUTH command. If it finds one, it runs the appropriate authentication protocol, and authentication either succeeds or fails. If there is no matching advertised mechanism, the AUTH command is rejected with a 504 error.

When a message is received from an authenticated host, the value of *$received_protocol* is set to “esmtpa” or “esmtpsa” instead of “esmtp” or “esmtps”, and *$sender_host_authenticated* contains the name (not the public name) of the authenticator driver that successfully authenticated the client from which the message was received. This variable is empty if there was no successful authentication.

33.4 Testing server authentication

Exim's **-bh** option can be useful for testing server authentication configurations. The data for the AUTH command has to be sent using base64 encoding. A quick way to produce such data for testing is the following Perl script:

```
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 on such data might be

```
encode '\0user\0password'
```

Note the use of single quotes to prevent the shell interpreting the backslashes, so that they can be interpreted by Perl to specify characters whose code value is zero.

Warning 1: If either of the user or password strings starts with an octal digit, you must use three zeros instead of one after the leading backslash. If you do not, the octal digit that starts your string will be incorrectly interpreted as part of the code for the first character.

Warning 2: If there are characters in the strings that Perl interprets specially, you must use a Perl escape to prevent them being misinterpreted. For example, a command such as

```
encode '\0user@domain.com\0pas$$word'
```

gives an incorrect answer because of the unescaped "@" and "$" characters.

If you have the **mimencode** command installed, another way to do produce base64-encoded strings is to run the command

```
echo -e -n `\0user\0password' | mimencode
```

The **-e** option of **echo** enables the interpretation of backslash escapes in the argument, and the **-n** option specifies no newline at the end of its output. However, not all versions of **echo** recognize these options, so you should check your version before relying on this suggestion.

33.5 Authentication by an Exim client

The *smtp* transport has two options called **hosts_require_auth** and **hosts_try_auth**. When the *smtp* transport connects to a server that announces support for authentication, and the host matches an entry in either of these options, Exim (as a client) tries to authenticate as follows:

- For each authenticator that is configured as a client, in the order in which they are defined in the configuration, 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, and Exim moves on to the next authenticator. Otherwise an expansion failure causes delivery to be deferred.
- If the result of the authentication attempt 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 and tries another one if possible. If all authentication attempts give permanent errors, or if there are no attempts because no mechanisms match (or option expansions force failure), what happens depends on whether the host matches **hosts_require_auth** or **hosts_try_auth**. In the first case, a temporary error is generated, and delivery is deferred. The error can be detected in the retry rules, and thereby turned into a permanent error if you wish. In the second case, Exim 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 the message came from a remote host, the authenticated sender is the one that was receiving on an incoming MAIL command, provided that the incoming connection was authenticated and the **server_mail_auth** condition allowed the authenticated sender to be retained. 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. However, if the **authenticated_sender** option is set on the *smtp* transport, it overrides the authenticated sender that was received with the message.

34. The plaintext authenticator

The *plaintext* authenticator can be configured to support the PLAIN and LOGIN authentication mechanisms, both of which transfer authentication data as plain (unencrypted) text (though base64 encoded). The use of plain text is a security risk; you are strongly advised to insist on the use of SMTP encryption (see chapter 39) if you use the PLAIN or LOGIN mechanisms. If you do use unencrypted plain text, you should not use the same passwords for SMTP connections as you do for login accounts.

34.1 Plaintext options

When configured as a server, *plaintext* uses the following options:

server_condition	Use: *authenticators*	Type: *string*†	Default: *unset*

This is actually a global authentication option, but it must be set in order to configure the *plaintext* driver as a server. Its use is described below.

server_prompts	Use: *plaintext*	Type: *string*†	Default: *unset*

The contents of this option, after expansion, must be a colon-separated list of prompt strings. If expansion fails, a temporary authentication rejection is given.

34.2 Using plaintext in a server

When running as a server, *plaintext* performs the authentication test by expanding a string. The data sent by the client with the AUTH command, or in response to subsequent prompts, is base64 encoded, and so may contain any byte values when decoded. If any data is supplied with the command, it is treated as a list of strings, separated by NULs (binary zeros), the first three of which are placed in the expansion variables *$auth1*, *$auth2*, and *$auth3* (neither LOGIN nor PLAIN uses more than three strings).

For compatibility with previous releases of Exim, the values are also placed in the expansion variables *$1*, *$2*, and *$3*. However, the use of these variables for this purpose is now deprecated, as it can lead to confusion in string expansions that also use them for other things.

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.

Once a sufficient number of data strings have been received, **server_condition** is expanded. If the expansion is forced to fail, authentication fails. Any other expansion failure 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 generic **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.

Warning: If you use a lookup in the expansion to find the user’s password, be sure to make the authentication fail if the user is unknown. There are good and bad examples at the end of the next section.

34.3 The PLAIN authentication mechanism

The PLAIN authentication mechanism (RFC 2595) specifies that three strings be sent as one item of data (that is, one combined string containing two NUL separators). The data is sent either as part of the AUTH command, or subsequently in response to an empty prompt from the server.

The second and third strings are a user name and a corresponding password. Using a single fixed user name and password as an example, this could be configured as follows:

```
fixed_plain:
  driver = plaintext
  public_name = PLAIN
  server_prompts = :
  server_condition = \
    ${if and {{eq{$auth2}{username}}{eq{$auth3}{mysecret}}}}
  server_set_id = $auth2
```

Note that the default result strings from **if** ("true" or an empty string) are exactly what we want here, so they need not be specified. Obviously, if the password contains expansion-significant characters such as dollar, backslash, or closing brace, they have to be escaped.

The **server_prompts** setting specifies a single, empty prompt (empty items at the end of a string list are ignored). If all the data comes as part of the AUTH command, as is commonly the case, the prompt is not used. This authenticator is advertised in the response to EHLO as

```
250-AUTH PLAIN
```

and a client host can authenticate itself by sending the command

```
AUTH PLAIN AHVzZXJuYW1lAG15c2VjcmV0
```

As this contains three strings (more than the number of prompts), no further data is required from the client. Alternatively, the client may just send

```
AUTH PLAIN
```

to initiate authentication, in which case the server replies with an empty prompt. The client must respond with the combined data string.

The data string is base64 encoded, as required by the RFC. This example, when decoded, is <*NUL*>`username`<*NUL*>`mysecret`, where <*NUL*> represents a zero byte. This is split up into three strings, the first of which is empty. The **server_condition** option in the authenticator checks that the second two are `username` and `mysecret` respectively.

Having just one fixed user name and password, as in this example, is not very realistic, though for a small organization with only a handful of authenticating clients it could make sense.

A more sophisticated instance of this authenticator could use the user name in *$auth2* to look up a password in a file or database, and maybe do an encrypted comparison (see **crypteq** in chapter 11). Here is a example of this approach, where the passwords are looked up in a DBM file. **Warning**: This is an incorrect example:

```
server_condition = \
  ${if eq{$auth3}{${lookup{$auth2}dbm{/etc/authpwd}}}}
```

The expansion uses the user name (*$auth2*) as the key to look up a password, which it then compares to the supplied password (*$auth3*). Why is this example incorrect? It works fine for existing users, but consider what happens if a non-existent user name is given. The lookup fails, but as no success/failure strings are given for the lookup, it yields an empty string. Thus, to defeat the authentication, all a client has to do is to supply a non-existent user name and an empty password. The correct way of writing this test is:

```
server_condition = ${lookup{$auth2}dbm{/etc/authpwd}\
  {${if eq{$value}{$auth3}}} {false}}
```

In this case, if the lookup succeeds, the result is checked; if the lookup fails, "false" is returned and authentication fails. If **crypteq** is being used instead of **eq**, the first example is in fact safe, because **crypteq** always fails if its second argument is empty. However, the second way of writing the test makes the logic clearer.

34.4 The LOGIN authentication mechanism

The LOGIN authentication mechanism is not documented in any RFC, but is in use in a number of programs. No data is sent with the AUTH command. Instead, a user name 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 = \
    ${if and {{eq{$auth1}{username}}{eq{$auth2}{mysecret}}}}
  server_set_id = $auth1
```

Because of the way plaintext operates, this authenticator accepts data supplied with the AUTH command (in contravention of the specification of LOGIN), but if the client does not supply it (as is the case for LOGIN clients), the prompt strings are used to obtain two data items.

Some clients are very particular about the precise text of the prompts. For example, Outlook Express is reported to recognize only "Username:" and "Password:". Here is an example of a LOGIN authenticator that uses those strings. It uses the **ldapauth** expansion condition to check the user name and password by binding to an LDAP server:

```
login:
  driver = plaintext
  public_name = LOGIN
  server_prompts = Username:: : Password::
  server_condition = ${if ldapauth \
    {user="cn=${quote_ldap_dn:$auth1},ou=people,o=example.org" \
    pass=${quote:$auth2} \
    ldap://ldap.example.org/}}
  server_set_id = uid=$auth1,ou=people,o=example.org
```

Note the use of the **quote_ldap_dn** operator to correctly quote the DN for authentication. However, the basic **quote** operator, rather than any of the LDAP quoting operators, is the correct one to use for the password, because quoting is needed only to make the password conform to the Exim syntax. At the LDAP level, the password is an uninterpreted string.

34.5 Support for different kinds of authentication

A number of string expansion features are provided for the purpose of interfacing to different ways of user authentication. These include checking traditionally encrypted passwords from */etc/passwd* (or equivalent), PAM, Radius, **ldapauth**, *pwcheck*, and *saslauthd*. For details see section 11.7.

34.6 Using plaintext in a client

The *plaintext* authenticator has two client options:

client_ignore_invalid_base64	Use: *plaintext*	Type: *boolean*	Default: *false*

If the client receives a server prompt that is not a valid base64 string, authentication is abandoned by default. However, if this option is set true, the error in the challenge is ignored and the client sends the response as usual.

client_send	Use: *plaintext*	Type: *string*†	Default: *unset*

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; any more strings are sent in response to prompts from the server. Before each string is expanded, the value of the most recent prompt is placed in the next *$auth<n>* variable, starting with *$auth1* for the first prompt. Up to three prompts are stored in this way. Thus, the prompt that is received in response to sending the first string (with the AUTH command) can be used in the expansion of the second string,

and so on. If an invalid base64 string is received when **client_ignore_invalid_base64** is set, an empty string is put in the *$auth<n>* variable.

Note: You cannot use expansion to create multiple strings, because splitting takes priority and happens first.

Because the PLAIN authentication mechanism requires NUL (binary zero) bytes in the data, 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 NULs. 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 the PLAIN authentication mechanism with a fixed user name and password:

```
fixed_plain:
  driver = plaintext
  public_name = PLAIN
  client_send = ^username^mysecret
```

The lack of colons means that the entire text is sent with the AUTH command, with the circumflex characters converted to NULs. A similar example that uses the LOGIN mechanism is:

```
fixed_login:
  driver = plaintext
  public_name = LOGIN
  client_send = : username : mysecret
```

The initial colon means that the first string is empty, so no data is sent with the AUTH command itself. The remaining strings are sent in response to prompts.

35. The cram_md5 authenticator

The CRAM-MD5 authentication mechanism is described in RFC 2195. The server sends a challenge string to the client, and the response consists of a user name and the CRAM-MD5 digest of the challenge string combined with a secret string (password) which is known to both server and client. Thus, the secret is not sent over the network as plain text, which makes this authenticator more secure than *plaintext*. However, the downside is that the secret has to be available in plain text at either end.

35.1 Using cram_md5 as a server

This authenticator has one server option, which must be set to configure the authenticator as a server:

server_secret	Use: *cram_md5*	Type: *string*†	Default: *unset*

When the server receives the client's response, the user name is placed in the expansion variable *$auth1*, 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 compatibility with previous releases of Exim, the user name is also placed in *$1*. However, the use of this variables for this purpose is now deprecated, as it can lead to confusion in string expansions that also use numeric variables for other things.

For example, the following authenticator checks that the user name given by the client is "ph10", and if so, uses "secret" as the password. For any other user name, authentication fails.

```
fixed_cram:
  driver = cram_md5
  public_name = CRAM-MD5
  server_secret = ${if eq{$auth1}{ph10}{secret}fail}
  server_set_id = $auth1
```

If authentication succeeds, the setting of **server_set_id** preserves the user name in *$authenticated_id*. A more typical configuration might look up the secret string in a file, using the user name as the key. For example:

```
lookup_cram:
  driver = cram_md5
  public_name = CRAM-MD5
  server_secret = ${lookup{$auth1}lsearch{/etc/authpwd}\
                  {$value}fail}
  server_set_id = $auth1
```

Note that this expansion explicitly forces failure if the lookup fails because *$auth1* contains an unknown user name.

35.2 Using cram_md5 as a client

When used as a client, the *cram_md5* authenticator has two options:

client_name	Use: *cram_md5*	Type: *string*†	Default: *the primary host name*

This string is expanded, and the result used as the user name data when computing the response to the server's challenge.

client_secret	Use: *cram_md5*	Type: *string*†	Default: *unset*

This option must be set for the authenticator to work as a client. Its value is expanded and the result used as the secret string when computing the response.

Different user names and secrets can be used for different servers by referring to *$host* or *$host_address* in the options. 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* authenticator, using fixed strings, is:

```
fixed_cram:
  driver = cram_md5
  public_name = CRAM-MD5
  client_name = ph10
  client_secret = secret
```

36. The cyrus_sasl authenticator

The code for this authenticator was provided by Matthew Byng-Maddick of A L Digital Ltd (**http://www.aldigital.co.uk**).

The *cyrus_sasl* authenticator provides server support for the Cyrus SASL library implementation of the RFC 2222 ("Simple Authentication and Security Layer"). This library supports a number of authentication mechanisms, including PLAIN and LOGIN, but also several others that Exim does not support directly. In particular, there is support for Kerberos authentication.

The *cyrus_sasl* authenticator provides a gatewaying mechanism directly to the Cyrus interface, so if your Cyrus library can do, for example, CRAM-MD5, then so can the *cyrus_sasl* authenticator. By default it uses the public name of the driver to determine which mechanism to support.

Where access to some kind of secret file is required, for example in GSSAPI or CRAM-MD5, it is worth noting that the authenticator runs as the Exim user, and that the Cyrus SASL library has no way of escalating privileges by default. You may also find you need to set environment variables, depending on the driver you are using.

The application name provided by Exim is "exim", so various SASL options may be set in *exim.conf* in your SASL directory. If you are using GSSAPI for Kerberos, note that because of limitations in the GSSAPI interface, changing the server keytab might need to be communicated down to the Kerberos layer independently. The mechanism for doing so is dependent upon the Kerberos implementation. For example, for Heimdal, the environment variable KRB5_KTNAME may be set to point to an alternative keytab file. Exim will pass this variable through from its own inherited environment when started as root or the Exim user. The keytab file needs to be readable by the Exim user.

36.1 Using cyrus_sasl as a server

The *cyrus_sasl* authenticator has four private options. It puts the username (on a successful authentication) into *$auth1*. For compatibility with previous releases of Exim, the username is also placed in *$1*. However, the use of this variable for this purpose is now deprecated, as it can lead to confusion in string expansions that also use numeric variables for other things.

server_hostname	Use: *cyrus_sasl*	Type: *string*†	Default: *see below*

This option selects the hostname that is used when communicating with the library. The default value is `$primary_hostname`. It is up to the underlying SASL plug-in what it does with this data.

server_mech	Use: *cyrus_sasl*	Type: *string*	Default: *see below*

This option selects the authentication mechanism this driver should use. The default is the value of the generic **public_name** option. This option allows you to use a different underlying mechanism from the advertised name. For example:

```
sasl:
  driver = cyrus_sasl
  public_name = X-ANYTHING
  server_mech = CRAM-MD5
  server_set_id = $auth1
```

server_realm	Use: *cyrus_sasl*	Type: *string*	Default: *unset*

This specifies the SASL realm that the server claims to be in.

server_service	Use: *cyrus_sasl*	Type: *string*	Default: `smtp`

This is the SASL service that the server claims to implement.

For straightforward cases, you do not need to set any of the authenticator's private options. All you need to do is to specify an appropriate mechanism as the public name. Thus, if you have a SASL library that supports CRAM-MD5 and PLAIN, you could have two authenticators as follows:

```
sasl_cram_md5:
  driver = cyrus_sasl
  public_name = CRAM-MD5
  server_set_id = $auth1

sasl_plain:
  driver = cyrus_sasl
  public_name = PLAIN
  server_set_id = $auth1
```

Cyrus SASL does implement the LOGIN authentication method, even though it is not a standard method. It is disabled by default in the source distribution, but it is present in many binary distributions.

37. The dovecot authenticator

This authenticator is an interface to the authentication facility of the Dovecot POP/IMAP server, which can support a number of authentication methods. If you are using Dovecot to authenticate POP/IMAP clients, it might be helpful to use the same mechanisms for SMTP authentication. This is a server authenticator only. There is only one option:

server_socket	Use: *dovecot*	Type: *string*	Default: *unset*

This option must specify the socket that is the interface to Dovecot authentication. The **public_name** option must specify an authentication mechanism that Dovecot is configured to support. You can have several authenticators for different mechanisms. For example:

```
dovecot_plain:
  driver = dovecot
  public_name = PLAIN
  server_socket = /var/run/dovecot/auth-client
  server_set_id = $auth1

dovecot_ntlm:
  driver = dovecot
  public_name = NTLM
  server_socket = /var/run/dovecot/auth-client
  server_set_id = $auth1
```

If the SMTP connection is encrypted, or if *$sender_host_address* is equal to *$received_ip_address* (that is, the connection is local), the "secured" option is passed in the Dovecot authentication command. If, for a TLS connection, a client certificate has been verified, the "valid-client-cert" option is passed. When authentication succeeds, the identity of the user who authenticated is placed in *$auth1*.

38. The spa authenticator

The *spa* authenticator provides client support for Microsoft's *Secure Password Authentication* mechanism, which is also sometimes known as NTLM (NT LanMan). The code for client side of this authenticator was contributed by Marc Prud'hommeaux, and much of it is taken from the Samba project (**http://www.samba.org**). The code for the server side was subsequently contributed by Tom Kistner. The mechanism works as follows:

- After the AUTH command has been accepted, the client sends an SPA authentication request based on the user name and optional domain.
- The server sends back a challenge.
- The client builds a challenge response which makes use of the user's password and sends it to the server, which then accepts or rejects it.

Encryption is used to protect the password in transit.

38.1 Using spa as a server

The *spa* authenticator has just one server option:

server_password	Use: *spa*	Type: *string*†	Default: *unset*

This option is expanded, and the result must be the cleartext password for the authenticating user, whose name is at this point in *$auth1*. For compatibility with previous releases of Exim, the user name is also placed in *$1*. However, the use of this variable for this purpose is now deprecated, as it can lead to confusion in string expansions that also use numeric variables for other things. For example:

```
spa:
  driver = spa
  public_name = NTLM
  server_password = \
    ${lookup{$auth1}lsearch{/etc/exim/spa_clearpass}{$value}fail}
```

If the expansion is forced to fail, authentication fails. Any other expansion failure causes a temporary error code to be returned.

38.2 Using spa as a client

The *spa* authenticator has the following client options:

client_domain	Use: *spa*	Type: *string*†	Default: *unset*

This option specifies an optional domain for the authentication.

client_password	Use: *spa*	Type: *string*†	Default: *unset*

This option specifies the user's password, and must be set.

client_username	Use: *spa*	Type: *string*†	Default: *unset*

This option specifies the user name, and must be set. Here is an example of a configuration of this authenticator for use with the mail servers at *msn.com*:

```
msn:
  driver = spa
```

```
public_name = MSN
client_username = msn/msn_username
client_password = msn_plaintext_password
client_domain = DOMAIN_OR_UNSET
```

39. Encrypted SMTP connections using TLS/SSL

Support for TLS (Transport Layer Security), formerly known as SSL (Secure Sockets Layer), is implemented by making use of the OpenSSL library or the GnuTLS library (Exim requires GnuTLS release 1.0 or later). There is no cryptographic code in the Exim distribution itself for implementing TLS. In order to use this feature you must install OpenSSL or GnuTLS, and then build a version of Exim that includes TLS support (see section 4.6). 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 certifi¤ cates are used.

RFC 3207 defines how SMTP connections can make use of encryption. Once a connection is established, the client issues a STARTTLS command. If the server accepts this, the client and the server negotiate an encryption mechanism. If the negotiation succeeds, the data that subsequently passes between them is encrypted.

Exim's ACLs can detect whether the current SMTP session is encrypted or not, and if so, what cipher suite is in use, whether the client supplied a certificate, and whether or not that certificate was verified. This makes it possible for an Exim server to deny or accept certain commands based on the encryption state.

Warning: Certain types of firewall and certain anti-virus products can disrupt TLS connections. You need to turn off SMTP scanning for these products in order to get TLS to work.

39.1 Support for the legacy "ssmtp" (aka "smtps") protocol

Early implementations of encrypted SMTP used a different TCP port from normal SMTP, and expected an encryption negotiation to start immediately, instead of waiting for a STARTTLS command from the client using the standard SMTP port. The protocol was called "ssmtp" or "smtps", and port 465 was allocated for this purpose.

This approach was abandoned when encrypted SMTP was standardized, but there are still some legacy clients that use it. Exim supports these clients by means of the **tls_on_connect_ports** global option. Its value must be a list of port numbers; the most common use is expected to be:

```
tls_on_connect_ports = 465
```

The port numbers specified by this option apply to all SMTP connections, both via the daemon and via *inetd*. You still need to specify all the ports that the daemon uses (by setting **daemon_smtp_ports** or **local_interfaces** or the **-oX** command line option) because **tls_on_connect_ports** does not add an extra port – rather, it specifies different behaviour on a port that is defined elsewhere.

There is also a **-tls-on-connect** command line option. This overrides **tls_on_connect_ports**; it forces the legacy behaviour for all ports.

39.2 OpenSSL vs GnuTLS

The first TLS support in Exim was implemented using OpenSSL. Support for GnuTLS followed later, when the first versions of GnuTLS were released. To build Exim to use GnuTLS, you need to set

```
USE_GNUTLS=yes
```

in Local/Makefile, in addition to

```
SUPPORT_TLS=yes
```

You must also set TLS_LIBS and TLS_INCLUDE appropriately, so that the include files and libraries for GnuTLS can be found.

There are some differences in usage when using GnuTLS instead of OpenSSL:

- The **tls_verify_certificates** option must contain the name of a file, not the name of a directory (for OpenSSL it can be either).

- The **tls_dhparam** option is ignored, because early versions of GnuTLS had no facility for varying its Diffie-Hellman parameters. I understand that this has changed, but Exim has not been updated to provide this facility.
- Distinguished Name (DN) strings reported by the OpenSSL library use a slash for separating fields; GnuTLS uses commas, in accordance with RFC 2253. This affects the value of the *$tls_peerdn* variable.
- OpenSSL identifies cipher suites using hyphens as separators, for example: DES-CBC3-SHA. GnuTLS uses underscores, for example: RSA_ARCFOUR_SHA. What is more, OpenSSL complains if underscores are present in a cipher list. To make life simpler, Exim changes underscores to hyphens for OpenSSL and hyphens to underscores for GnuTLS when processing lists of cipher suites in the **tls_require_ciphers** options (the global option and the *smtp* transport option).
- The **tls_require_ciphers** options operate differently, as described in the sections 39.4 and 39.5.

39.3 GnuTLS parameter computation

GnuTLS uses RSA and D-H parameters that may take a substantial amount of time to compute. It is unreasonable to re-compute them for every TLS session. Therefore, Exim keeps this data in a file in its spool directory, called *gnutls-params*. The file is owned by the Exim user and is readable only by its owner. Every Exim process that start up GnuTLS reads the RSA and D-H parameters from this file. If the file does not exist, the first Exim process that needs it computes the data and writes it to a temporary file which is renamed once it is complete. It does not matter if several Exim processes do this simultaneously (apart from wasting a few resources). Once a file is in place, new Exim processes immediately start using it.

For maximum security, the parameters that are stored in this file should be recalculated periodically, the frequency depending on your paranoia level. Arranging this is easy in principle; just delete the file when you want new values to be computed. However, there may be a problem. The calculation of new parameters needs random numbers, and these are obtained from */dev/random*. If the system is not very active, */dev/random* may delay returning data until enough randomness (entropy) is available. This may cause Exim to hang for a substantial amount of time, causing timeouts on incoming connections.

The solution is to generate the parameters externally to Exim. They are stored in *gnutls-params* in PEM format, which means that they can be generated externally using the *certtool* command that is part of GnuTLS.

To replace the parameters with new ones, instead of deleting the file and letting Exim re-create it, you can generate new parameters using *certtool* and, when this has been done, replace Exim's cache file by renaming. The relevant commands are something like this:

```
# rm -f new-params
# touch new-params
# chown exim:exim new-params
# chmod 0400 new-params
# certtool --generate-privkey --bits 512 >new-params
# echo "" >>new-params
# certtool --generate-dh-params --bits 1024 >> new-params
# mv new-params gnutls-params
```

If Exim never has to generate the parameters itself, the possibility of stalling is removed.

39.4 Requiring specific ciphers in OpenSSL

There is a function in the OpenSSL library that can be passed a list of cipher suites before the cipher negotiation takes place. This specifies which ciphers are acceptable. The list is colon separated and may contain names like DES-CBC3-SHA. Exim passes the expanded value of **tls_require_ciphers** directly to this function call. The following quotation from the OpenSSL documentation specifies what forms of item are allowed in the cipher string:

- It can consist of a single cipher suite such as RC4-SHA.
- It can represent a list of cipher suites containing a certain algorithm, or cipher suites of a certain type. For example SHA1 represents all ciphers suites using the digest algorithm SHA1 and SSLv3 represents all SSL v3 algorithms.
- Lists of cipher suites can be combined in a single cipher string using the + character. This is used as a logical and operation. For example SHA1+DES represents all cipher suites containing the SHA1 and the DES algorithms.

Each cipher string can be optionally preceded by one of the characters `!`, − or +.

- If `!` is used, the ciphers are permanently deleted from the list. The ciphers deleted can never reappear in the list even if they are explicitly stated.
- If − is used, the ciphers are deleted from the list, but some or all of the ciphers can be added again by later options.
- If + is used, the ciphers are moved to the end of the list. This option does not add any new ciphers; it just moves matching existing ones.

If none of these characters is present, the string is interpreted as a list of ciphers to be appended to the current preference list. If the list includes any ciphers already present they will be ignored: that is, they will not be moved to the end of the list.

39.5 Requiring specific ciphers or other parameters in GnuTLS

The GnuTLS library allows the caller to specify separate lists of permitted key exchange methods, main cipher algorithms, MAC algorithms, and protocols. Unfortunately, these lists are numerical, and the library does not have a function for turning names into numbers. Consequently, lists of recognized names have to be built into the application. The permitted key exchange methods, ciphers, and MAC algorithms may be used in any combination to form a cipher suite. This is unlike OpenSSL, where complete cipher suite names are passed to its control function.

For compatibility with OpenSSL, the **tls_require_ciphers** option can be set to complete cipher suite names such as RSA_ARCFOUR_SHA, but for GnuTLS this option controls only the cipher algorithms. Exim searches each item in the list for the name of an available algorithm. For example, if the list contains RSA_AES_SHA, then AES is recognized, and the behaviour is exactly the same as if just AES were given.

There are additional options called **gnutls_require_kx**, **gnutls_require_mac**, and **gnutls_require_protocols** that can be used to restrict the key exchange methods, MAC algorithms, and protocols, respectively. These options are ignored if OpenSSL is in use.

All four options are available as global options, controlling how Exim behaves as a server, and also as options of the *smtp* transport, controlling how Exim behaves as a client. All the values are string expanded. After expansion, the values must be colon-separated lists, though the separator can be changed in the usual way.

Each of the four lists starts out with a default set of algorithms. If the first item in a list does *not* start with an exclamation mark, all the default items are deleted. In this case, only those that are explicitly specified can be used. If the first item in a list *does* start with an exclamation mark, the defaults are left on the list.

Then, any item that starts with an exclamation mark causes the relevant entry to be removed from the list, and any item that does not start with an exclamation mark causes a new entry to be added to the list. Unrecognized items in the list are ignored. Thus:

```
tls_require_ciphers = !ARCFOUR
```

allows all the defaults except ARCFOUR, whereas

```
tls_require_ciphers = AES : 3DES
```

allows only cipher suites that use AES or 3DES.

For **tls_require_ciphers** the recognized names are AES_256, AES_128, AES (both of the preceding), 3DES, ARCFOUR_128, ARCFOUR_40, and ARCFOUR (both of the preceding). The default list does not contain all of these; it just has AES_256, AES_128, 3DES, and ARCFOUR_128.

For **gnutls_require_kx**, the recognized names are DHE_RSA, RSA (which includes DHE_RSA), DHE_DSS, and DHE (which includes both DHE_RSA and DHE_DSS). The default list contains RSA, DHE_DSS, DHE_RSA.

For **gnutls_require_mac**, the recognized names are SHA (synonym SHA1), and MD5. The default list contains SHA, MD5.

For **gnutls_require_protocols**, the recognized names are TLS1 and SSL3. The default list contains TLS1, SSL3.

In a server, the order of items in these lists is unimportant. The server advertises the availability of all the relevant cipher suites. However, in a client, the order in the **tls_require_ciphers** list specifies a preference order for the cipher algorithms. The first one in the client's list that is also advertised by the server is tried first. The default order is as listed above.

39.6 Configuring an Exim server to use TLS

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 you need to set some other options in order to make TLS available, and also it is sensible for systems that want to use TLS only as a client.

If a client issues a STARTTLS command and there is some configuration problem in the server, the command is rejected with a 454 error. If the client persists in trying to issue SMTP commands, all except QUIT are rejected with the error

```
554 Security failure
```

If a STARTTLS command is issued within an existing TLS session, it is rejected with a 554 error code.

To enable TLS operations on a server, you must set **tls_advertise_hosts** to match some hosts. You can, of course, set it to * to match all hosts. However, this is not all you need to do. TLS sessions to a server won't work without some further configuration at the server end.

It is rumoured that all existing clients that support TLS/SSL use RSA encryption. To make this work you need to set, in the server,

```
tls_certificate = /some/file/name
tls_privatekey = /some/file/name
```

These options are, in fact, expanded strings, so you can make them depend on the identity of the client that is connected if you wish. 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, and must always be given as full path names. They can be the same file if both the certificate and the key are contained within it. If **tls_privatekey** is not set, or if its expansion is forced to fail or results in an empty string, this is assumed to be the case. The certificate file may also contain intermediate certifi¤ cates that need to be sent to the client to enable it to authenticate the server's certificate.

If you do not understand about certificates and keys, please try to find a source of this background information, which is not Exim-specific. (There are a few comments below in section 39.11.)

Note: These options do not apply when Exim is operating as a client – they apply only in the case of a server. If you need to use a certificate in an Exim client, you must set the options of the same names in an *smtp* transport.

With just these options, an Exim server will be able to use TLS. It does not require the client to have a certificate (but see below for how to insist on this). There is one other option that may be needed in other situations. If

```
tls_dhparam = /some/file/name
```

is set, the SSL library is initialized for the use of Diffie-Hellman ciphers with the parameters contained in the file. This increases the set of cipher suites that the server supports. See the command

```
openssl dhparam
```

for a way of generating this data. At present, **tls_dhparam** is used only when Exim is linked with OpenSSL. It is ignored if GnuTLS is being used.

The strings supplied for these three options 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.

The variable *$tls_cipher* is set to the cipher suite that was negotiated for an incoming TLS connection. It is included in the *Received:* header of an incoming message (by default – you can, of course, change this), and it is also included in the log line that records a message's arrival, keyed by "X=", unless the **tls_cipher** log selector is turned off. The **encrypted** condition can be used to test for specific cipher suites in ACLs. (For outgoing SMTP deliveries, *$tls_cipher* is reset – see section 39.9.)

Once TLS has been established, the ACLs that run for subsequent SMTP commands can check the name of the cipher suite and vary their actions accordingly. The cipher suite names vary, depending on which TLS library is being used. For example, OpenSSL uses the name DES-CBC3-SHA for the cipher suite which in other contexts is known as TLS_RSA_WITH_3DES_EDE_CBC_SHA. Check the OpenSSL or GnuTLS documentation for more details.

39.7 Requesting and verifying client certificates

If you want an Exim server to request a certificate when negotiating a TLS session with a client, you must set either **tls_verify_hosts** or **tls_try_verify_hosts**. You can, of course, set either of them to * to apply to all TLS connections. For any host that matches one of these options, Exim requests a certificate as part of the setup of the TLS session. The contents of the certificate are verified by comparing it with a list of expected certificates. These must be available in a file or, for OpenSSL only (not GnuTLS), a directory, identified by **tls_verify_certificates**.

A file can contain multiple certificates, concatenated end to end. If a directory is used (OpenSSL only), 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 a single certificate.

The difference between **tls_verify_hosts** and **tls_try_verify_hosts** is what happens if the client does not supply a certificate, or if the certificate does not match any of the certificates in the collection named by **tls_verify_certificates**. If the client matches **tls_verify_hosts**, the attempt to set up a TLS session is aborted, and the incoming connection is dropped. If the client matches **tls_try_verify_hosts**, the (encrypted) SMTP session continues. ACLs that run for subsequent SMTP commands can detect the fact that no certificate was verified, and vary their actions accordingly. For example, you can insist on a certificate before accepting a message for relaying, but not when the message is destined for local delivery.

When a client supplies a certificate (whether it verifies or not), the value of the Distinguished Name of the certificate 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 *Received:* header by default. You can arrange for it to be logged, keyed by "DN=", by setting the **tls_peerdn** log selector, and you can use **received_header_text** to change the *Received:* header. When no certificate is supplied, *$tls_peerdn* is empty.

39.8 Revoked certificates

Certificate issuing authorities issue Certificate Revocation Lists (CRLs) when certificates are revoked. If you have such a list, you can pass it to an Exim server using the global option called **tls_crl** and to an Exim client using an identically named option for the *smtp* transport. In each case, the value of the option is expanded and must then be the name of a file that contains a CRL in PEM format.

39.9 Configuring an Exim client to use TLS

The **tls_cipher** and **tls_peerdn** log selectors apply to outgoing SMTP deliveries as well as to incoming, the latter one causing logging of the server certificate's DN. 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 Exim is built with TLS support, and TLS is advertised by a server, the *smtp* transport always tries 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 you do not want Exim to attempt to send messages unencrypted when an attempt to set up an encrypted connection fails in any way, you can set **hosts_require_tls** to a list of hosts for which encryption is mandatory. For those hosts, delivery is always deferred if an encrypted connection cannot be set up. If there are any other hosts for the address, they are tried in the usual way.

When the server host is not in **hosts_require_tls**, Exim may try to deliver the message unencrypted. It always does this if the response to STARTTLS is a 5*xx* code. For a temporary error code, or for a failure to negotiate a TLS session after a success response code, what happens is controlled by the **tls_tempfail_tryclear** option of the *smtp* transport. If it is false, delivery to this host is deferred, and other hosts (if available) are tried. If it is true, Exim attempts to deliver unencrypted after a 4*xx* response to STARTTLS, and if STARTTLS is accepted, but the subsequent TLS negotiation fails, Exim closes the current connection (because it is in an unknown state), opens a new one to the same host, and then tries the delivery unencrypted.

The **tls_certificate** and **tls_privatekey** options of the *smtp* transport provide the client with a certifi¤ cate, which is passed to the server if it requests it. If the server is Exim, it will request a certificate only if **tls_verify_hosts** or **tls_try_verify_hosts** matches the client. **Note**: These options must be set in the *smtp* transport for Exim to use TLS when it is operating as a client. Exim does not assume that a server certificate (set by the global options of the same name) should also be used when operating as a client.

If **tls_verify_certificates** is set, it must name a file or, for OpenSSL only (not GnuTLS), a directory, that contains a collection of expected server certificates. The client verifies the server's certificate against this collection, taking into account any revoked certificates that are in the list defined by **tls_ crl**.

If **tls_require_ciphers** is set on the *smtp* transport, it must contain a list of permitted cipher suites. If either of these checks fails, delivery to the current host is abandoned, and the *smtp* transport tries to deliver to alternative hosts, if any.

All the TLS options in the *smtp* transport are expanded before use, with *$host* and *$host_address* containing the name and 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.

Before an SMTP connection is established, the *$tls_cipher* and *$tls_peerdn* variables are emptied. (Until the first connection, they contain the values that were set when the message was received.) If STARTTLS is subsequently successfully obeyed, these variables are set to the relevant values for the outgoing connection.

39.10 Multiple messages on the same encrypted TCP/IP connection

Exim sends multiple messages down the same TCP/IP connection by starting up an entirely new delivery process for each message, passing the socket from one process to the next. This implementation does not fit well with the use of TLS, because there is quite a lot of state information associated

with a TLS connection, not just a socket identification. Passing all the state information to a new process is not feasible. Consequently, Exim shuts down an existing TLS session before passing the socket to a new process. The new process may then try to start a new TLS session, and if successful, may try to re-authenticate if AUTH is in use, before sending the next message.

The RFC is not clear as to whether or not an SMTP session continues in clear after TLS has been shut down, or whether TLS may be restarted again later, as just described. However, if the server is Exim, this shutdown and reinitialization works. It is not known which (if any) other servers operate successfully if the client closes a TLS session and continues with unencrypted SMTP, but there are certainly some that do not work. For such servers, Exim should not pass the socket to another process, because the failure of the subsequent attempt to use it would cause Exim to record a temporary host error, and delay other deliveries to that host.

To test for this case, Exim sends an EHLO command to the server after closing down the TLS session. If this fails in any way, the connection is closed instead of being passed to a new delivery process, but no retry information is recorded.

There is also a manual override; you can set **hosts_nopass_tls** on the *smtp* transport to match those hosts for which Exim should not pass connections to new processes if TLS has been used.

39.11 Certificates and all that

In order to understand fully how TLS works, you need to know about certificates, certificate signing, and certificate authorities. This is not the place to give a tutorial, especially as I do not know very much about it myself. Some helpful introduction can be found in the FAQ for the SSL addition to Apache, currently at

http://www.modssl.org/docs/2.7/ssl_faq.html#ToC24

Other parts of the *modssl* documentation are also helpful, and have links to further files. Eric Rescorla's book, *SSL and TLS*, published by Addison-Wesley (ISBN 0-201-61598-3), contains both introductory and more in-depth descriptions. Some sample programs taken from the book are available from

http://www.rtfm.com/openssl-examples/

39.12 Certificate chains

The file named by **tls_certificate** may contain more than one certificate. This is useful in the case where the certificate that is being sent is validated by an intermediate certificate which the other end does not have. Multiple certificates must be in the correct order in the file. First the host's certificate itself, then the first intermediate certificate to validate the issuer of the host certificate, then the next intermediate certificate to validate the issuer of the first intermediate certificate, and so on, until finally (optionally) the root certificate. The root certificate must already be trusted by the recipient for validation to succeed, of course, but if it's not preinstalled, sending the root certificate along with the rest makes it available for the user to install if the receiving end is a client MUA that can interact with a user.

39.13 Self-signed certificates

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. The **-nodes** option is important: if you do not set it, the key is encrypted with a passphrase that you are prompted for, and any use that is made of the key causes more prompting for the passphrase. 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.

However, many clients require that the certificate presented by the server be a user (also called "leaf" or "site") certificate, and not a self-signed certificate. In this situation, the self-signed certificate described above must be installed on the client host as a trusted root *certification authority* (CA), and the certificate used by Exim must be a user certificate signed with that self-signed certificate.

For information on creating self-signed CA certificates and using them to sign user certificates, see the *General implementation overview* chapter of the Open-source PKI book, available online at **http://ospkibook.sourceforge.net/**.

40. Access control lists

Access Control Lists (ACLs) are defined in a separate section of the run time configuration file, headed by "begin acl". Each ACL definition starts with a name, terminated by a colon. Here is a complete ACL section that contains just one very small ACL:

```
begin acl
small_acl:
  accept   hosts = one.host.only
```

You can have as many lists as you like in the ACL section, and the order in which they appear does not matter. The lists are self-terminating.

The majority of ACLs are used to control Exim's behaviour when it receives certain SMTP commands. This applies both to incoming TCP/IP connections, and when a local process submits a message using SMTP by specifying the **-bs** option. The most common use is for controlling which recipients are accepted in incoming messages. In addition, you can define an ACL that is used to check local non-SMTP messages. The default configuration file contains an example of a realistic ACL for checking RCPT commands. This is discussed in chapter 7.

40.1 Testing ACLs

The **-bh** command line option provides a way of testing your ACL configuration locally by running a fake SMTP session with which you interact. The host *relay-test.mail-abuse.org* provides a service for checking your relaying configuration (see section 40.49 for more details).

40.2 Specifying when ACLs are used

In order to cause an ACL to be used, you have to name it in one of the relevant options in the main part of the configuration. These options are:

acl_not_smtp	ACL for non-SMTP messages
acl_not_smtp_mime	ACL for non-SMTP MIME parts
acl_not_smtp_start	ACL at start of non-SMTP message
acl_smtp_auth	ACL for AUTH
acl_smtp_connect	ACL for start of SMTP connection
acl_smtp_data	ACL after DATA is complete
acl_smtp_etrn	ACL for ETRN
acl_smtp_expn	ACL for EXPN
acl_smtp_helo	ACL for HELO or EHLO
acl_smtp_mail	ACL for MAIL
acl_smtp_mailauth	ACL for the AUTH parameter of MAIL
acl_smtp_mime	ACL for content-scanning MIME parts
acl_smtp_notquit	ACL for non-QUIT terminations
acl_smtp_predata	ACL at start of DATA command
acl_smtp_quit	ACL for QUIT
acl_smtp_rcpt	ACL for RCPT
acl_smtp_starttls	ACL for STARTTLS
acl_smtp_vrfy	ACL for VRFY

For example, if you set

```
acl_smtp_rcpt = small_acl
```

the little ACL defined above is used whenever Exim receives a RCPT command in an SMTP dialogue. The majority of policy tests on incoming messages can be done when RCPT commands arrive. A rejection of RCPT should cause the sending MTA to give up on the recipient address contained in the RCPT command, whereas rejection at other times may cause the client MTA to keep on trying to deliver the message. It is therefore recommended that you do as much testing as possible at RCPT time.

40.3 The non-SMTP ACLs

The non-SMTP ACLs apply to all non-interactive incoming messages, that is, they apply to batched SMTP as well as to non-SMTP messages. (Batched SMTP is not really SMTP.) Many of the ACL conditions (for example, host tests, and tests on the state of the SMTP connection such as encryption and authentication) are not relevant and are forbidden in these ACLs. However, the sender and recipients are known, so the **senders** and **sender_domains** conditions and the *$sender_address* and *$recipients* variables can be used. Variables such as *$authenticated_sender* are also available. You can specify added header lines in any of these ACLs.

The **acl_not_smtp_start** ACL is run right at the start of receiving a non-SMTP message, before any of the message has been read. (This is the analogue of the **acl_smtp_predata** ACL for SMTP input.) In the case of batched SMTP input, it runs after the DATA command has been reached. The result of this ACL is ignored; it cannot be used to reject a message. If you really need to, you could set a value in an ACL variable here and reject based on that in the **acl_not_smtp** ACL. However, this ACL can be used to set controls, and in particular, it can be used to set

```
control = suppress_local_fixups
```

This cannot be used in the other non-SMTP ACLs because by the time they are run, it is too late.

The **acl_not_smtp_mime** ACL is available only when Exim is compiled with the content-scanning extension. For details, see chapter 41.

The **acl_not_smtp** ACL is run just before the *local_scan()* function. Any kind of rejection is treated as permanent, because there is no way of sending a temporary error for these kinds of message.

40.4 The SMTP connect ACL

The ACL test specified by **acl_smtp_connect** happens at the start of an SMTP session, after the test specified by **host_reject_connection** (which is now an anomaly) and any TCP Wrappers testing (if configured). If the connection is accepted by an **accept** verb that has a **message** modifier, the contents of the message override the banner message that is otherwise specified by the **smtp_banner** option.

40.5 The EHLO/HELO ACL

The ACL test specified by **acl_smtp_helo** happens when the client issues an EHLO or HELO command, after the tests specified by **helo_accept_junk_hosts**, **helo_allow_chars**, **helo_verify_hosts**, and **helo_try_verify_hosts**. Note that a client may issue more than one EHLO or HELO command in an SMTP session, and indeed is required to issue a new EHLO or HELO after successfully setting up encryption following a STARTTLS command.

If the command is accepted by an **accept** verb that has a **message** modifier, the message may not contain more than one line (it will be truncated at the first newline and a panic logged if it does). Such a message cannot affect the EHLO options that are listed on the second and subsequent lines of an EHLO response.

40.6 The DATA ACLs

Two ACLs are associated with the DATA command, because it is two-stage command, with two responses being sent to the client. When the DATA command is received, the ACL defined by **acl_smtp_predata** is obeyed. This gives you control after all the RCPT commands, but before the message itself is received. It offers the opportunity to give a negative response to the DATA command before the data is transmitted. Header lines added by MAIL or RCPT ACLs are not visible at this time, but any that are defined here are visible when the **acl_smtp_data** ACL is run.

You cannot test the contents of the message, for example, to verify addresses in the headers, at RCPT time or when the DATA command is received. Such tests have to appear in the ACL that is run after the message itself has been received, before the final response to the DATA command is sent. This is the ACL specified by **acl_smtp_data**, which is the second ACL that is associated with the DATA command.

For both of these ACLs, it is not possible to reject individual recipients. An error response rejects the entire message. Unfortunately, it is known that some MTAs do not treat hard (5*xx*) responses to the DATA command (either before or after the data) correctly – they keep the message on their queues and try again later, but that is their problem, though it does waste some of your resources.

40.7 The SMTP MIME ACL

The **acl_smtp_mime** option is available only when Exim is compiled with the content-scanning extension. For details, see chapter 41.

40.8 The QUIT ACL

The ACL for the SMTP QUIT command is anomalous, in that the outcome of the ACL does not affect the response code to QUIT, which is always 221. Thus, the ACL does not in fact control any access. For this reason, the only verbs that are permitted are **accept** and **warn**.

This ACL can be used for tasks such as custom logging at the end of an SMTP session. For example, you can use ACL variables in other ACLs to count messages, recipients, etc., and log the totals at QUIT time using one or more **logwrite** modifiers on a **warn** verb.

Warning: Only the *$acl_cx* variables can be used for this, because the *$acl_mx* variables are reset at the end of each incoming message.

You do not need to have a final **accept**, but if you do, you can use a **message** modifier to specify custom text that is sent as part of the 221 response to QUIT.

This ACL is run only for a "normal" QUIT. For certain kinds of disastrous failure (for example, failure to open a log file, or when Exim is bombing out because it has detected an unrecoverable error), all SMTP commands from the client are given temporary error responses until QUIT is received or the connection is closed. In these special cases, the QUIT ACL does not run.

40.9 The not-QUIT ACL

The not-QUIT ACL, specified by **smtp_notquit_acl**, is run in most cases when an SMTP session ends without sending QUIT. However, when Exim itself is is bad trouble, such as being unable to write to its log files, this ACL is not run, because it might try to do things (such as write to log files) that make the situation even worse.

Like the QUIT ACL, this ACL is provided to make it possible to do customized logging or to gather statistics, and its outcome is ignored. The **delay** modifier is forbidden in this ACL, and the only permitted verbs are **accept** and **warn**.

When the not-QUIT ACL is running, the variable *$smtp_notquit_reason* is set to a string that indicates the reason for the termination of the SMTP connection. The possible values are:

`acl-drop`	Another ACL issued a **drop** command
`bad-commands`	Too many unknown or non-mail commands
`command-timeout`	Timeout while reading SMTP commands
`connection-lost`	The SMTP connection has been lost
`data-timeout`	Timeout while reading message data
`local-scan-error`	The *local_scan()* function crashed
`local-scan-timeout`	The *local_scan()* function timed out
`signal-exit`	SIGTERM or SIGINT
`synchronization-error`	SMTP synchronization error
`tls-failed`	TLS failed to start

In most cases when an SMTP connection is closed without having received QUIT, Exim sends an SMTP response message before actually closing the connection. With the exception of the `acl-drop` case, the default message can be overridden by the **message** modifier in the not-QUIT ACL. In the case of a **drop** verb in another ACL, it is the message from the other ACL that is used.

40.10 Finding an ACL to use

The value of an **acl_smtp_***xxx* option is expanded before use, so you can use different ACLs in different circumstances. For example,

```
acl_smtp_rcpt = ${if ={25}{$interface_port} \
                     {acl_check_rcpt} {acl_check_rcpt_submit} }
```

In the default configuration file there are some example settings for providing an RFC 4409 message submission service on port 587 and a non-standard "smtps" service on port 465. You can use a string expansion like this to choose an ACL for MUAs on these ports which is more appropriate for this purpose than the default ACL on port 25.

The expanded string does not have to be the name of an ACL in the configuration file; there are other possibilities. Having expanded the string, Exim searches for an ACL as follows:

- If the string begins with a slash, Exim uses it as a file name, and reads its contents as an ACL. The lines are processed in the same way as lines in the Exim configuration file. In particular, continuation lines are supported, blank lines are ignored, as are lines whose first non-whitespace character is "#". If the file does not exist or cannot be read, an error occurs (typically causing a temporary failure of whatever caused the ACL to be run). For example:

  ```
  acl_smtp_data = /etc/acls/\
    ${lookup{$sender_host_address}lsearch\
    {/etc/acllist}{$value}{default}}
  ```

  This looks up an ACL file to use on the basis of the host's IP address, falling back to a default if the lookup fails. If an ACL is successfully read from a file, it is retained in memory for the duration of the Exim process, so that it can be re-used without having to re-read the file.

- If the string does not start with a slash, and does not contain any spaces, Exim searches the ACL section of the configuration for an ACL whose name matches the string.

- If no named ACL is found, or if the string contains spaces, Exim parses the string as an inline ACL. This can save typing in cases where you just want to have something like

  ```
  acl_smtp_vrfy = accept
  ```

  in order to allow free use of the VRFY command. Such a string may contain newlines; it is processed in the same way as an ACL that is read from a file.

40.11 ACL return codes

Except for the QUIT ACL, which does not affect the SMTP return code (see section 40.8 above), the result of running an ACL is either "accept" or "deny", or, if some test cannot be completed (for example, if a database is down), "defer". These results cause 2*xx*, 5*xx*, and 4*xx* return codes, respectively, to be used in the SMTP dialogue. A fourth return, "error", occurs when there is an error such as invalid syntax in the ACL. This also causes a 4*xx* return code.

For the non-SMTP ACL, "defer" and "error" are treated in the same way as "deny", because there is no mechanism for passing temporary errors to the submitters of non-SMTP messages.

ACLs that are relevant to message reception may also return "discard". This has the effect of "accept", but causes either the entire message or an individual recipient address to be discarded. In other words, it is a blackholing facility. Use it with care.

If the ACL for MAIL returns "discard", all recipients are discarded, and no ACL is run for subsequent RCPT commands. The effect of "discard" in a RCPT ACL is to discard just the one recipient address. If there are no recipients left when the message's data is received, the DATA ACL is not run. A "discard" return from the DATA or the non-SMTP ACL discards all the remaining recipients. The "discard" return is not permitted for the **acl_smtp_predata** ACL.

The *local_scan()* function is always run, even if there are no remaining recipients; it may create new recipients.

40.12 Unset ACL options

The default actions when any of the **acl_*xxx*** options are unset are not all the same. **Note**: These defaults apply only when the relevant ACL is not defined at all. For any defined ACL, the default action when control reaches the end of the ACL statements is "deny".

For **acl_smtp_quit** and **acl_not_smtp_start** there is no default because these two are ACLs that are used only for their side effects. They cannot be used to accept or reject anything.

For **acl_not_smtp**, **acl_smtp_auth**, **acl_smtp_connect**, **acl_smtp_data**, **acl_smtp_helo**, **acl_smtp_mail**, **acl_smtp_mailauth**, **acl_smtp_mime**, **acl_smtp_predata**, and **acl_smtp_starttls**, the action when the ACL is not defined is "accept".

For the others (**acl_smtp_etrn**, **acl_smtp_expn**, **acl_smtp_rcpt**, and **acl_smtp_vrfy**), the action when the ACL is not defined is "deny". This means that **acl_smtp_rcpt** must be defined in order to receive any messages over an SMTP connection. For an example, see the ACL in the default configuration file.

40.13 Data for message ACLs

When a MAIL or RCPT ACL, or either of the DATA ACLs, is running, the variables that contain information about the host and the message's sender (for example, *$sender_host_address* and *$sender_address*) are set, and can be used in ACL statements. In the case of RCPT (but not MAIL or DATA), *$domain* and *$local_part* are set from the argument address. The entire SMTP command is available in *$smtp_command*.

When an ACL for the AUTH parameter of MAIL is running, the variables that contain information about the host are set, but *$sender_address* is not yet set. Section 33.2 contains a discussion of this parameter and how it is used.

The *$message_size* variable is set to the value of the SIZE parameter on the MAIL command at MAIL, RCPT and pre-data time, or to -1 if that parameter is not given. The value is updated to the true message size by the time the final DATA ACL is run (after the message data has been received).

The *$rcpt_count* variable increases by one for each RCPT command received. The *$recipients_count* variable increases by one each time a RCPT command is accepted, so while an ACL for RCPT is being processed, it contains the number of previously accepted recipients. At DATA time (for both the DATA ACLs), *$rcpt_count* contains the total number of RCPT commands, and *$recipients_count* contains the total number of accepted recipients.

40.14 Data for non-message ACLs

When an ACL is being run for AUTH, EHLO, ETRN, EXPN, HELO, STARTTLS, or VRFY, the remainder of the SMTP command line is placed in *$smtp_command_argument*, and the entire SMTP command is available in *$smtp_command*. These variables can be tested using a **condition** condition. For example, here is an ACL for use with AUTH, which insists that either the session is encrypted, or the CRAM-MD5 authentication method is used. In other words, it does not permit authentication methods that use cleartext passwords on unencrypted connections.

```
acl_check_auth:
  accept encrypted = *
  accept condition = ${if eq{${uc:$smtp_command_argument}}\
                     {CRAM-MD5}}
  deny   message   = TLS encryption or CRAM-MD5 required
```

(Another way of applying this restriction is to arrange for the authenticators that use cleartext passwords not to be advertised when the connection is not encrypted. You can use the generic **server_advertise_condition** authenticator option to do this.)

40.15 Format of an ACL

An individual ACL consists of a number of statements. Each statement starts with a verb, optionally followed by a number of conditions and "modifiers". Modifiers can change the way the verb operates, define error and log messages, set variables, insert delays, and vary the processing of accepted messages.

If all the conditions are met, the verb is obeyed. The same condition may be used (with different arguments) more than once in the same statement. This provides a means of specifying an "and" conjunction between conditions. For example:

```
deny  dnslists = list1.example
dnslists = list2.example
```

If there are no conditions, the verb is always obeyed. Exim stops evaluating the conditions and modifiers when it reaches a condition that fails. What happens then depends on the verb (and in one case, on a special modifier). Not all the conditions make sense at every testing point. For example, you cannot test a sender address in the ACL that is run for a VRFY command.

40.16 ACL verbs

The ACL verbs are as follows:

- **accept**: If all the conditions are met, the ACL returns "accept". If any of the conditions are not met, what happens depends on whether **endpass** appears among the conditions (for syntax see below). If the failing condition is before **endpass**, control is passed to the next ACL statement; if it is after **endpass**, the ACL returns "deny". Consider this statement, used to check a RCPT command:

  ```
  accept domains = +local_domains
  endpass
  verify = recipient
  ```

  If the recipient domain does not match the **domains** condition, control passes to the next statement. If it does match, the recipient is verified, and the command is accepted if verification succeeds. However, if verification fails, the ACL yields "deny", because the failing condition is after **endpass**.

  The **endpass** feature has turned out to be confusing to many people, so its use is not recommended nowadays. It is always possible to rewrite an ACL so that **endpass** is not needed, and it is no longer used in the default configuration.

  If a **message** modifier appears on an **accept** statement, its action depends on whether or not **endpass** is present. In the absence of **endpass** (when an **accept** verb either accepts or passes control to the next statement), **message** can be used to vary the message that is sent when an SMTP command is accepted. For example, in a RCPT ACL you could have:

  ```
  accept  <some conditions>
          message = OK, I will allow you through today
  ```

  You can specify an SMTP response code, optionally followed by an "extended response code" at the start of the message, but the first digit must be the same as would be sent by default, which is 2 for an **accept** verb.

  If **endpass** is present in an **accept** statement, **message** specifies an error message that is used when access is denied. This behaviour is retained for backward compatibility, but current "best practice" is to avoid the use of **endpass**.

- **defer**: If all the conditions are true, the ACL returns "defer" which, in an SMTP session, causes a 4*xx* response to be given. For a non-SMTP ACL, **defer** is the same as **deny**, because there is no way of sending a temporary error. For a RCPT command, **defer** is much the same as using a *redirect* router and `:defer:` while verifying, but the **defer** verb can be used in any ACL, and even for a recipient it might be a simpler approach.

- **deny**: If all the conditions are met, the ACL returns "deny". If any of the conditions are not met, control is passed to the next ACL statement. For example,

```
deny dnslists = blackholes.mail-abuse.org
```

rejects commands from hosts that are on a DNS black list.

- **discard**: This verb behaves like **accept**, except that it returns "discard" from the ACL instead of "accept". It is permitted only on ACLs that are concerned with receiving messages. When all the conditions are true, the sending entity receives a "success" response. However, **discard** causes recipients to be discarded. If it is used in an ACL for RCPT, just the one recipient is discarded; if used for MAIL, DATA or in the non-SMTP ACL, all the message's recipients are discarded. Recipients that are discarded before DATA do not appear in the log line when the **log_recipients** log selector is set.

  If the **log_message** modifier is set when **discard** operates, its contents are added to the line that is automatically written to the log. The **message** modifier operates exactly as it does for **accept**.

- **drop**: This verb behaves like **deny**, except that an SMTP connection is forcibly closed after the 5*xx* error message has been sent. For example:

  ```
  drop   message   = I don't take more than 20 RCPTs
         condition = ${if > {$rcpt_count}{20}}
  ```

  There is no difference between **deny** and **drop** for the connect-time ACL. The connection is always dropped after sending a 550 response.

- **require**: If all the conditions are met, control is passed to the next ACL statement. If any of the conditions are not met, the ACL returns "deny". For example, when checking a RCPT command,

  ```
  require message = Sender did not verify
          verify  = sender
  ```

  passes control to subsequent statements only if the message's sender can be verified. Otherwise, it rejects the command. Note the positioning of the **message** modifier, before the **verify** condition. The reason for this is discussed in section 40.18.

- **warn**: If all the conditions are true, a line specified by the **log_message** modifier is written to Exim's main log. Control always passes to the next ACL statement. If any condition is false, the log line is not written. If an identical log line is requested several times in the same message, only one copy is actually written to the log. If you want to force duplicates to be written, use the **logwrite** modifier instead.

  If **log_message** is not present, a **warn** verb just checks its conditions and obeys any "immediate" modifiers (such as **control**, **set**, **logwrite**, and **add_header**) that appear before the first failing condition. There is more about adding header lines in section 40.22.

  If any condition on a **warn** statement cannot be completed (that is, there is some sort of defer), the log line specified by **log_message** is not written. This does not include the case of a forced failure from a lookup, which is considered to be a successful completion. After a defer, no further conditions or modifiers in the **warn** statement are processed. The incident is logged, and the ACL continues to be processed, from the next statement onwards.

  When one of the **warn** conditions is an address verification that fails, the text of the verification failure message is in *$acl_verify_message*. If you want this logged, you must set it up explicitly. For example:

  ```
  warn   !verify = sender
         log_message = sender verify failed: $acl_verify_message
  ```

At the end of each ACL there is an implicit unconditional **deny**.

As you can see from the examples above, the conditions and modifiers are written one to a line, with the first one on the same line as the verb, and subsequent ones on following lines. If you have a very long condition, you can continue it onto several physical lines by the usual backslash continuation mechanism. It is conventional to align the conditions vertically.

40.17 ACL variables

There are some special variables that can be set during ACL processing. They can be used to pass information between different ACLs, different invocations of the same ACL in the same SMTP connection, and between ACLs and the routers, transports, and filters that are used to deliver a message. The names of these variables must begin with *$acl_c* or *$acl_m*, followed either by a digit or an underscore, but the remainder of the name can be any sequence of alphanumeric characters and underscores that you choose. There is no limit on the number of ACL variables. The two sets act as follows:

- The values of those variables whose names begin with *$acl_c* persist throughout an SMTP connection. They are never reset. Thus, a value that is set while receiving one message is still available when receiving the next message on the same SMTP connection.
- The values of those variables whose names begin with *$acl_m* persist only while a message is being received. They are reset afterwards. They are also reset by MAIL, RSET, EHLO, HELO, and after starting up a TLS session.

When a message is accepted, the current values of all the ACL variables are preserved with the message and are subsequently made available at delivery time. The ACL variables are set by a modifier called **set**. For example:

```
accept hosts = whatever
       set acl_m4 = some value
accept authenticated = *
       set acl_c_auth = yes
```

Note: A leading dollar sign is not used when naming a variable that is to be set. If you want to set a variable without taking any action, you can use a **warn** verb without any other modifiers or conditions.

What happens if a syntactically valid but undefined ACL variable is referenced depends on the setting of the **strict_acl_vars** option. If it is false (the default), an empty string is substituted; if it is true, an error is generated.

Versions of Exim before 4.64 have a limited set of numbered variables, but their names are compatible, so there is no problem with upgrading.

40.18 Condition and modifier processing

An exclamation mark preceding a condition negates its result. For example:

```
deny   domains = *.dom.example
      !verify  = recipient
```

causes the ACL to return "deny" if the recipient domain ends in *dom.example* and the recipient address cannot be verified. Sometimes negation can be used on the right-hand side of a condition. For example, these two statements are equivalent:

```
deny  hosts = !192.168.3.4
deny !hosts =  192.168.3.4
```

However, for many conditions (**verify** being a good example), only left-hand side negation of the whole condition is possible.

The arguments of conditions and modifiers are expanded. A forced failure of an expansion causes a condition to be ignored, that is, it behaves as if the condition is true. Consider these two statements:

```
accept  senders = ${lookup{$host_name}lsearch\
                  {/some/file}{$value}fail}
accept  senders = ${lookup{$host_name}lsearch\
                  {/some/file}{$value}{}}
```

Each attempts to look up a list of acceptable senders. If the lookup succeeds, the returned list is searched, but if the lookup fails the behaviour is different in the two cases. The **fail** in the first

statement causes the condition to be ignored, leaving no further conditions. The **accept** verb therefore succeeds. The second statement, however, generates an empty list when the lookup fails. No sender can match an empty list, so the condition fails, and therefore the **accept** also fails.

ACL modifiers appear mixed in with conditions in ACL statements. Some of them specify actions that are taken as the conditions for a statement are checked; others specify text for messages that are used when access is denied or a warning is generated. The **control** modifier affects the way an incoming message is handled.

The positioning of the modifiers in an ACL statement important, because the processing of a verb ceases as soon as its outcome is known. Only those modifiers that have already been encountered will take effect. For example, consider this use of the **message** modifier:

```
require message = Can't verify sender
        verify  = sender
        message = Can't verify recipient
        verify  = recipient
        message = This message cannot be used
```

If sender verification fails, Exim knows that the result of the statement is "deny", so it goes no further. The first **message** modifier has been seen, so its text is used as the error message. If sender verifi¤ cation succeeds, but recipient verification fails, the second message is used. If recipient verification succeeds, the third message becomes "current", but is never used because there are no more conditions to cause failure.

For the **deny** verb, on the other hand, it is always the last **message** modifier that is used, because all the conditions must be true for rejection to happen. Specifying more than one **message** modifier does not make sense, and the message can even be specified after all the conditions. For example:

```
deny   hosts = ...
      !senders = *@my.domain.example
       message = Invalid sender from client host
```

The "deny" result does not happen until the end of the statement is reached, by which time Exim has set up the message.

40.19 ACL modifiers

The ACL modifiers are as follows:

add_header = <*text*>
: This modifier specifies one or more header lines that are to be added to an incoming message, assuming, of course, that the message is ultimately accepted. For details, see section 40.22.

continue = <*text*>
: This modifier does nothing of itself, and processing of the ACL always continues with the next condition or modifier. The value of **continue** is in the side effects of expanding its argument. Typically this could be used to update a database. It is really just a syntactic tidiness, to avoid having to write rather ugly lines like this:

  ```
  condition = ${if eq{0}{<some expansion>}{true}{true}}
  ```

  Instead, all you need is

  ```
  continue = <some expansion>
  ```

control = <*text*>
: This modifier affects the subsequent processing of the SMTP connection or of an incoming message that is accepted. The effect of the first type of control lasts for the duration of the connection, whereas the effect of the second type lasts only until the current message has been received. The message-specific controls always apply to the whole message, not to individual recipients, even if the **control** modifier appears in a RCPT ACL.

  As there are now quite a few controls that can be applied, they are described separately in section 40.20. The **control** modifier can be used in several different ways. For example:

- It can be at the end of an **accept** statement:

```
accept  ...some conditions
        control = queue_only
```

  In this case, the control is applied when this statement yields "accept", in other words, when the conditions are all true.

- It can be in the middle of an **accept** statement:

```
accept  ...some conditions...
        control = queue_only
        ...some more conditions...
```

  If the first set of conditions are true, the control is applied, even if the statement does not accept because one of the second set of conditions is false. In this case, some subsequent statement must yield "accept" for the control to be relevant.

- It can be used with **warn** to apply the control, leaving the decision about accepting or denying to a subsequent verb. For example:

```
warn    ...some conditions...
        control = freeze
accept  ...
```

  This example of **warn** does not contain **message**, **log_message**, or **logwrite**, so it does not add anything to the message and does not write a log entry.

- If you want to apply a control unconditionally, you can use it with a **require** verb. For example:

```
require  control = no_multiline_responses
```

delay = <*time*>

This modifier may appear in any ACL. It causes Exim to wait for the time interval before proceeding. However, when testing Exim using the **-bh** option, the delay is not actually imposed (an appropriate message is output instead). The time is given in the usual Exim notation, and the delay happens as soon as the modifier is processed. In an SMTP session, pending output is flushed before the delay is imposed.

Like **control**, **delay** can be used with **accept** or **deny**, for example:

```
deny    ...some conditions...
        delay = 30s
```

The delay happens if all the conditions are true, before the statement returns "deny". Compare this with:

```
deny    delay = 30s
        ...some conditions...
```

which waits for 30s before processing the conditions. The **delay** modifier can also be used with **warn** and together with **control**:

```
warn    ...some conditions...
        delay = 2m
        control = freeze
accept  ...
```

If **delay** is encountered when the SMTP PIPELINING extension is in use, responses to several commands are no longer buffered and sent in one packet (as they would normally be) because all output is flushed before imposing the delay. This optimization is disabled so that a number of small delays do not appear to the client as one large aggregated delay that might provoke an unwanted timeout. You can, however, disable output flushing for **delay** by using a **control** modifier to set **no_delay_flush**.

endpass
This modifier, which has no argument, is recognized only in **accept** and **discard** statements. It marks the boundary between the conditions whose failure causes control to pass to the next statement, and the conditions whose failure causes the ACL to return "deny". This concept has proved to be confusing to some people, so the use of **endpass** is no longer recommended as "best practice". See the description of **accept** above for more details.

log_message = <*text*>
This modifier sets up a message that is used as part of the log message if the ACL denies access or a **warn** statement's conditions are true. For example:

```
require log_message = wrong cipher suite $tls_cipher
        encrypted   = DES-CBC3-SHA
```

log_message is also used when recipients are discarded by **discard**. For example:

```
discard <some conditions>
        log_message = Discarded $local_part@$domain because...
```

When access is denied, **log_message** adds to any underlying error message that may exist because of a condition failure. For example, while verifying a recipient address, a *:fail:* redirection might have already set up a message.

The message may be defined before the conditions to which it applies, because the string expansion does not happen until Exim decides that access is to be denied. This means that any variables that are set by the condition are available for inclusion in the message. For example, the *$dnslist_<xxx>* variables are set after a DNS black list lookup succeeds. If the expansion of **log_message** fails, or if the result is an empty string, the modifier is ignored.

If you want to use a **warn** statement to log the result of an address verification, you can use *$acl_verify_message* to include the verification error message.

If **log_message** is used with a **warn** statement, "Warning:" is added to the start of the logged message. If the same warning log message is requested more than once while receiving a single email message, only one copy is actually logged. If you want to log multiple copies, use **logwrite** instead of **log_message**. In the absence of **log_message** and **logwrite**, nothing is logged for a successful **warn** statement.

If **log_message** is not present and there is no underlying error message (for example, from the failure of address verification), but **message** is present, the **message** text is used for logging rejections. However, if any text for logging contains newlines, only the first line is logged. In the absence of both **log_message** and **message**, a default built-in message is used for logging rejections.

log_reject_target = <*log name list*>
This modifier makes it possible to specify which logs are used for messages about ACL rejections. Its argument is a colon-separated list of words that can be "main", "reject", or "panic". The default is `main:reject`. The list may be empty, in which case a rejection is not logged at all. For example, this ACL fragment writes no logging information when access is denied:

```
deny <some conditions>
     log_reject_target =
```

This modifier can be used in SMTP and non-SMTP ACLs. It applies to both permanent and temporary rejections.

logwrite = <*text*>
This modifier writes a message to a log file as soon as it is encountered when processing an ACL. (Compare **log_message**, which, except in the case of **warn** and **discard**, is used only if the ACL statement denies access.) The **logwrite** modifier can be used to log special incidents in ACLs. For example:

```
accept <some special conditions>
       control  = freeze
       logwrite = froze message because ...
```

By default, the message is written to the main log. However, it may begin with a colon, followed by a comma-separated list of log names, and then another colon, to specify exactly which logs are to be written. For example:

```
logwrite = :main,reject: text for main and reject logs
logwrite = :panic: text for panic log only
```

message = <*text*>
This modifier sets up a text string that is expanded and used as a response message when an ACL statement terminates the ACL with an "accept", "deny", or "defer" response. (In the case of the **accept** and **discard** verbs, there is some complication if **endpass** is involved; see the description of **accept** for details.)

The expansion of the message happens at the time Exim decides that the ACL is to end, not at the time it processes **message**. If the expansion fails, or generates an empty string, the modifier is ignored. Here is an example where **message** must be specified first, because the ACL ends with a rejection if the **hosts** condition fails:

```
require  message = Host not recognized
         hosts = 10.0.0.0/8
```

(Once a condition has failed, no further conditions or modifiers are processed.)

For ACLs that are triggered by SMTP commands, the message is returned as part of the SMTP response. The use of **message** with **accept** (or **discard**) is meaningful only for SMTP, as no message is returned when a non-SMTP message is accepted. In the case of the connect ACL, accepting with a message modifier overrides the value of **smtp_banner**. For the EHLO/HELO ACL, a customized accept message may not contain more than one line (otherwise it will be truncated at the first newline and a panic logged), and it cannot affect the EHLO options.

When SMTP is involved, the message may begin with an overriding response code, consisting of three digits optionally followed by an "extended response code" of the form *n.n.n*, each code being followed by a space. For example:

```
deny  message = 599 1.2.3 Host not welcome
      hosts = 192.168.34.0/24
```

The first digit of the supplied response code must be the same as would be sent by default. A panic occurs if it is not. Exim uses a 550 code when it denies access, but for the predata ACL, note that the default success code is 354, not 2*xx*.

Notwithstanding the previous paragraph, for the QUIT ACL, unlike the others, the message modifier cannot override the 221 response code.

The text in a **message** modifier is literal; any quotes are taken as literals, but because the string is expanded, backslash escapes are processed anyway. If the message contains newlines, this gives rise to a multi-line SMTP response.

If **message** is used on a statement that verifies an address, the message specified overrides any message that is generated by the verification process. However, the original message is available in the variable *$acl_verify_message*, so you can incorporate it into your message if you wish. In particular, if you want the text from **:fail:** items in *redirect* routers to be passed back as part of the SMTP response, you should either not use a **message** modifier, or make use of *$acl_verify_message*.

For compatibility with previous releases of Exim, a **message** modifier that is used with a **warn** verb behaves in a similar way to the **add_header** modifier, but this usage is now deprecated. However, **message** acts only when all the conditions are true, wherever it appears in an ACL command, whereas **add_header** acts as soon as it is encountered. If **message** is used with **warn** in an ACL that is not concerned with receiving a message, it has no effect.

set <*acl_name*> = <*value*>
This modifier puts a value into one of the ACL variables (see section 40.17).

40.20 Use of the control modifier

The **control** modifier supports the following settings:

control = allow_auth_unadvertised
This modifier allows a client host to use the SMTP AUTH command even when it has not been advertised in response to EHLO. Furthermore, because there are apparently some really broken clients that do this, Exim will accept AUTH after HELO (rather than EHLO) when this control is set. It should be used only if you really need it, and you should limit its use to those broken clients that do not work without it. For example:

```
warn hosts   = 192.168.34.25
     control = allow_auth_unadvertised
```

Normally, when an Exim server receives an AUTH command, it checks the name of the authentication mechanism that is given in the command to ensure that it matches an advertised mechanism. When this control is set, the check that a mechanism has been advertised is bypassed. Any configured mechanism can be used by the client. This control is permitted only in the connection and HELO ACLs.

control = caseful_local_part
control = caselower_local_part
These two controls are permitted only in the ACL specified by **acl_smtp_rcpt** (that is, during RCPT processing). By default, the contents of *$local_part* are lower cased before ACL processing. If "caseful_local_part" is specified, any uppercase letters in the original local part are restored in *$local_part* for the rest of the ACL, or until a control that sets "caselower_local_part" is encountered.

These controls affect only the current recipient. Moreover, they apply only to local part handling that takes place directly in the ACL (for example, as a key in lookups). If a test to verify the recipient is obeyed, the case-related handling of the local part during the verification is controlled by the router configuration (see the **caseful_local_part** generic router option).

This facility could be used, for example, to add a spam score to local parts containing upper case letters. For example, using *$acl_m4* to accumulate the spam score:

```
warn  control = caseful_local_part
      set acl_m4 = ${eval:\
                    $acl_m4 + \
                    ${if match{$local_part}{[A-Z]}{1}{0}}\
                   }
      control = caselower_local_part
```

Notice that we put back the lower cased version afterwards, assuming that is what is wanted for subsequent tests.

control = enforce_sync
control = no_enforce_sync
These controls make it possible to be selective about when SMTP synchronization is enforced. The global option **smtp_enforce_sync** specifies the initial state of the switch (it is true by default). See the description of this option in chapter 14 for details of SMTP synchronization checking.

The effect of these two controls lasts for the remainder of the SMTP connection. They can appear in any ACL except the one for the non-SMTP messages. The most straightforward place to put them is in the ACL defined by **acl_smtp_connect**, which is run at the start of an incoming SMTP connection, before the first synchronization check. The expected use is to turn off the synchronization checks for badly-behaved hosts that you nevertheless need to work with.

control = fakedefer/<*message*>
This control works in exactly the same way as **fakereject** (described below) except that it causes an SMTP 450 response after the message data instead of a 550 response. You must take care when using **fakedefer** because it causes the messages to be duplicated when the sender retries. Therefore, you should not use **fakedefer** if the message is to be delivered normally.

control = fakereject/<*message*>

This control is permitted only for the MAIL, RCPT, and DATA ACLs, in other words, only when an SMTP message is being received. If Exim accepts the message, instead the final 250 response, a 550 rejection message is sent. However, Exim proceeds to deliver the message as normal. The control applies only to the current message, not to any subsequent ones that may be received in the same SMTP connection.

The text for the 550 response is taken from the **control** modifier. If no message is supplied, the following is used:

```
550-Your message has been rejected but is being
550-kept for evaluation.
550-If it was a legitimate message, it may still be
550 delivered to the target recipient(s).
```

This facility should be used with extreme caution.

control = freeze

This control is permitted only for the MAIL, RCPT, DATA, and non-SMTP ACLs, in other words, only when a message is being received. If the message is accepted, it is placed on Exim's queue and frozen. The control applies only to the current message, not to any subsequent ones that may be received in the same SMTP connection.

This modifier can optionally be followed by `/no_tell`. If the global option **freeze_tell** is set, it is ignored for the current message (that is, nobody is told about the freezing), provided all the **control=freeze** modifiers that are obeyed for the current message have the `/no_tell` option.

control = no_delay_flush

Exim normally flushes SMTP output before implementing a delay in an ACL, to avoid unexpected timeouts in clients when the SMTP PIPELINING extension is in use. This control, as long as it is encountered before the **delay** modifier, disables such output flushing.

control = no_callout_flush

Exim normally flushes SMTP output before performing a callout in an ACL, to avoid unexpected timeouts in clients when the SMTP PIPELINING extension is in use. This control, as long as it is encountered before the **verify** condition that causes the callout, disables such output flushing.

control = no_mbox_unspool

This control is available when Exim is compiled with the content scanning extension. Content scanning may require a copy of the current message, or parts of it, to be written in "mbox format" to a spool file, for passing to a virus or spam scanner. Normally, such copies are deleted when they are no longer needed. If this control is set, the copies are not deleted. The control applies only to the current message, not to any subsequent ones that may be received in the same SMTP connection. It is provided for debugging purposes and is unlikely to be useful in production.

control = no_multiline_responses

This control is permitted for any ACL except the one for non-SMTP messages. It seems that there are broken clients in use that cannot handle multiline SMTP responses, despite the fact that RFC 821 defined them over 20 years ago.

If this control is set, multiline SMTP responses from ACL rejections are suppressed. One way of doing this would have been to put out these responses as one long line. However, RFC 2821 specifies a maximum of 512 bytes per response ("use multiline responses for more" it says – ha!), and some of the responses might get close to that. So this facility, which is after all only a sop to broken clients, is implemented by doing two very easy things:

- Extra information that is normally output as part of a rejection caused by sender verification failure is omitted. Only the final line (typically "sender verification failed") is sent.
- If a **message** modifier supplies a multiline response, only the first line is output.

The setting of the switch can, of course, be made conditional on the calling host. Its effect lasts until the end of the SMTP connection.

control = no_pipelining
This control turns off the advertising of the PIPELINING extension to SMTP in the current session. To be useful, it must be obeyed before Exim sends its response to an EHLO command. Therefore, it should normally appear in an ACL controlled by **acl_smtp_connect** or **acl_smtp_helo**. See also **pipelining_advertise_hosts**.

control = queue_only
This control is permitted only for the MAIL, RCPT, DATA, and non-SMTP ACLs, in other words, only when a message is being received. If the message is accepted, it is placed on Exim's queue and left there for delivery by a subsequent queue runner. No immediate delivery process is started. In other words, it has the effect as the **queue_only** global option. However, the control applies only to the current message, not to any subsequent ones that may be received in the same SMTP connection.

control = submission/<*options*>
This control is permitted only for the MAIL, RCPT, and start of data ACLs (the latter is the one defined by **acl_smtp_predata**). Setting it tells Exim that the current message is a submission from a local MUA. In this case, Exim operates in "submission mode", and applies certain fixups to the message if necessary. For example, it add a *Date:* header line if one is not present. This control is not permitted in the **acl_smtp_data** ACL, because that is too late (the message has already been created).

Chapter 44 describes the processing that Exim applies to messages. Section 44.1 covers the processing that happens in submission mode; the available options for this control are described there. The control applies only to the current message, not to any subsequent ones that may be received in the same SMTP connection.

control = suppress_local_fixups
This control applies to locally submitted (non TCP/IP) messages, and is the complement of `control = submission`. It disables the fixups that are normally applied to locally-submitted messages. Specifically:

- Any *Sender:* header line is left alone (in this respect, it is a dynamic version of **local_sender_retain**).
- No *Message-ID:*, *From:*, or *Date:* header lines are added.
- There is no check that *From:* corresponds to the actual sender.

This control may be useful when a remotely-originated message is accepted, passed to some scanning program, and then re-submitted for delivery. It can be used only in the **acl_smtp_mail**, **acl_smtp_rcpt**, **acl_smtp_predata**, and **acl_not_smtp_start** ACLs, because it has to be set before the message's data is read.

Note: This control applies only to the current message, not to any others that are being submitted at the same time using **-bs** or **-bS**.

40.21 Summary of message fixup control

All four possibilities for message fixups can be specified:

- Locally submitted, fixups applied: the default.
- Locally submitted, no fixups applied: use `control = suppress_local_fixups`.
- Remotely submitted, no fixups applied: the default.
- Remotely submitted, fixups applied: use `control = submission`.

40.22 Adding header lines in ACLs

The **add_header** modifier can be used to add one or more extra header lines to an incoming message, as in this example:

```
warn dnslists = sbl.spamhaus.org : \
                dialup.mail-abuse.org
     add_header = X-blacklisted-at: $dnslist_domain
```

The **add_header** modifier is permitted in the MAIL, RCPT, PREDATA, DATA, MIME, and non-SMTP ACLs (in other words, those that are concerned with receiving a message). The message must ultimately be accepted for **add_header** to have any significant effect. You can use **add_header** with any ACL verb, including **deny** (though this is potentially useful only in a RCPT ACL).

If the data for the **add_header** modifier contains one or more newlines that are not followed by a space or a tab, it is assumed to contain multiple header lines. Each one is checked for valid syntax; `X-ACL-Warn:` is added to the front of any line that is not a valid header line.

Added header lines are accumulated during the MAIL, RCPT, and predata ACLs. They are added to the message before processing the DATA and MIME ACLs. However, if an identical header line is requested more than once, only one copy is actually added to the message. Further header lines may be accumulated during the DATA and MIME ACLs, after which they are added to the message, again with duplicates suppressed. Thus, it is possible to add two identical header lines to an SMTP message, but only if one is added before DATA and one after. In the case of non-SMTP messages, new headers are accumulated during the non-SMTP ACLs, and are added to the message after all the ACLs have run. If a message is rejected after DATA or by the non-SMTP ACL, all added header lines are included in the entry that is written to the reject log.

Header lines are not visible in string expansions until they are added to the message. It follows that header lines defined in the MAIL, RCPT, and predata ACLs are not visible until the DATA ACL and MIME ACLs are run. Similarly, header lines that are added by the DATA or MIME ACLs are not visible in those ACLs. Because of this restriction, you cannot use header lines as a way of passing data between (for example) the MAIL and RCPT ACLs. If you want to do this, you can use ACL variables, as described in section 40.17.

The **add_header** modifier acts immediately it is encountered during the processing of an ACL. Notice the difference between these two cases:

```
accept add_header = ADDED: some text
       <some condition>

accept <some condition>
       add_header = ADDED: some text
```

In the first case, the header line is always added, whether or not the condition is true. In the second case, the header line is added only if the condition is true. Multiple occurrences of **add_header** may occur in the same ACL statement. All those that are encountered before a condition fails are honoured.

For compatibility with previous versions of Exim, a **message** modifier for a **warn** verb acts in the same way as **add_header**, except that it takes effect only if all the conditions are true, even if it appears before some of them. Furthermore, only the last occurrence of **message** is honoured. This usage of **message** is now deprecated. If both **add_header** and **message** are present on a **warn** verb, both are processed according to their specifications.

By default, new header lines are added to a message at the end of the existing header lines. However, you can specify that any particular header line should be added right at the start (before all the *Received:* lines), immediately after the first block of *Received:* lines, or immediately before any line that is not a *Received:* or *Resent-something:* header.

This is done by specifying ":at_start:", ":after_received:", or ":at_start_rfc:" (or, for completeness, ":at_end:") before the text of the header line, respectively. (Header text cannot start with a colon, as there has to be a header name first.) For example:

```
warn add_header = \
       :after_received:X-My-Header: something or other...
```

If more than one header line is supplied in a single **add_header** modifier, each one is treated independently and can therefore be placed differently. If you add more than one line at the start, or after the Received: block, they end up in reverse order.

Warning: This facility currently applies only to header lines that are added in an ACL. It does NOT work for header lines that are added in a system filter or in a router or transport.

40.23 ACL conditions

Some of conditions listed in this section are available only when Exim is compiled with the content-scanning extension. They are included here briefly for completeness. More detailed descriptions can be found in the discussion on content scanning in chapter 41.

Not all conditions are relevant in all circumstances. For example, testing senders and recipients does not make sense in an ACL that is being run as the result of the arrival of an ETRN command, and checks on message headers can be done only in the ACLs specified by **acl_smtp_data** and **acl_not_smtp**. You can use the same condition (with different parameters) more than once in the same ACL statement. This provides a way of specifying an "and" conjunction. The conditions are as follows:

acl = <*name of acl or ACL string or file name* >
The possible values of the argument are the same as for the **acl_smtp_***xxx* options. The named or inline ACL is run. If it returns "accept" the condition is true; if it returns "deny" the condition is false. If it returns "defer", the current ACL returns "defer" unless the condition is on a **warn** verb. In that case, a "defer" return makes the condition false. This means that further processing of the **warn** verb ceases, but processing of the ACL continues.

If the nested **acl** returns "drop" and the outer condition denies access, the connection is dropped. If it returns "discard", the verb must be **accept** or **discard**, and the action is taken immediately – no further conditions are tested.

ACLs may be nested up to 20 deep; the limit exists purely to catch runaway loops. This condition allows you to use different ACLs in different circumstances. For example, different ACLs can be used to handle RCPT commands for different local users or different local domains.

authenticated = <*string list*>
If the SMTP connection is not authenticated, the condition is false. Otherwise, the name of the authenticator is tested against the list. To test for authentication by any authenticator, you can set

```
authenticated = *
```

condition = <*string*>
This feature allows you to make up custom conditions. If the result of expanding the string is an empty string, the number zero, or one of the strings "no" or "false", the condition is false. If the result is any non-zero number, or one of the strings "yes" or "true", the condition is true. For any other value, some error is assumed to have occurred, and the ACL returns "defer". However, if the expansion is forced to fail, the condition is ignored. The effect is to treat it as true, whether it is positive or negative.

decode = <*location*>
This condition is available only when Exim is compiled with the content-scanning extension, and it is allowed only in the ACL defined by **acl_smtp_mime**. It causes the current MIME part to be decoded into a file. If all goes well, the condition is true. It is false only if there are problems such as a syntax error or a memory shortage. For more details, see chapter 41.

demime = <*extension list*>
This condition is available only when Exim is compiled with the content-scanning extension. Its use is described in section 41.6.

dnslists = <*list of domain names and other data*>
This condition checks for entries in DNS black lists. These are also known as "RBL lists", after the original Realtime Blackhole List, but note that the use of the lists at *mail-abuse.org* now carries a charge. There are too many different variants of this condition to describe briefly here. See sections 40.24–40.34 for details.

domains = <*domain list*>
This condition is relevant only after a RCPT command. It checks that the domain of the recipient address is in the domain list. If percent-hack processing is enabled, it is done before this test is done. If the check succeeds with a lookup, the result of the lookup is placed in *$domain_data* until the next **domains** test.

Note carefully (because many people seem to fall foul of this): you cannot use **domains** in a DATA ACL.

encrypted = <*string list*>
If the SMTP connection is not encrypted, the condition is false. Otherwise, the name of the cipher suite in use is tested against the list. To test for encryption without testing for any specific cipher suite(s), set

```
encrypted = *
```

hosts = < *host list*>
This condition tests that the calling host matches the host list. If you have name lookups or wildcarded host names and IP addresses in the same host list, you should normally put the IP addresses first. For example, you could have:

```
accept hosts = 10.9.8.7 : dbm;/etc/friendly/hosts
```

The lookup in this example uses the host name for its key. This is implied by the lookup type "dbm". (For a host address lookup you would use "net-dbm" and it wouldn't matter which way round you had these two items.)

The reason for the problem with host names lies in the left-to-right way that Exim processes lists. It can test IP addresses without doing any DNS lookups, but when it reaches an item that requires a host name, it fails if it cannot find a host name to compare with the pattern. If the above list is given in the opposite order, the **accept** statement fails for a host whose name cannot be found, even if its IP address is 10.9.8.7.

If you really do want to do the name check first, and still recognize the IP address even if the name lookup fails, you can rewrite the ACL like this:

```
accept hosts = dbm;/etc/friendly/hosts
accept hosts = 10.9.8.7
```

The default action on failing to find the host name is to assume that the host is not in the list, so the first **accept** statement fails. The second statement can then check the IP address.

If a **hosts** condition is satisfied by means of a lookup, the result of the lookup is made available in the *$host_data* variable. This allows you, for example, to set up a statement like this:

```
deny  hosts = net-lsearch;/some/file
message = $host_data
```

which gives a custom error message for each denied host.

local_parts = <*local part list*>
This condition is relevant only after a RCPT command. It checks that the local part of the recipient address is in the list. If percent-hack processing is enabled, it is done before this test. If the check succeeds with a lookup, the result of the lookup is placed in *$local_part_data*, which remains set until the next **local_parts** test.

malware = <*option*>
This condition is available only when Exim is compiled with the content-scanning extension. It causes the incoming message to be scanned for viruses. For details, see chapter 41.

mime_regex = <*list of regular expressions*>
This condition is available only when Exim is compiled with the content-scanning extension, and it is allowed only in the ACL defined by **acl_smtp_mime**. It causes the current MIME part to be scanned for a match with any of the regular expressions. For details, see chapter 41.

ratelimit = <*parameters*>
This condition can be used to limit the rate at which a user or host submits messages. Details are given in section 40.35.

recipients = <*address list*>
This condition is relevant only after a RCPT command. It checks the entire recipient address against a list of recipients.

regex = <*list of regular expressions*>
This condition is available only when Exim is compiled with the content-scanning extension, and is available only in the DATA, MIME, and non-SMTP ACLs. It causes the incoming message to be scanned for a match with any of the regular expressions. For details, see chapter 41.

sender_domains = <*domain list*>
This condition tests the domain of the sender of the message against the given domain list. **Note**: The domain of the sender address is in *$sender_address_domain*. It is *not* put in *$domain* during the testing of this condition. This is an exception to the general rule for testing domain lists. It is done this way so that, if this condition is used in an ACL for a RCPT command, the recipient's domain (which is in *$domain*) can be used to influence the sender checking.

Warning: It is a bad idea to use this condition on its own as a control on relaying, because sender addresses are easily, and commonly, forged.

senders = <*address list*>
This condition tests the sender of the message against the given list. To test for a bounce message, which has an empty sender, set

```
senders = :
```

Warning: It is a bad idea to use this condition on its own as a control on relaying, because sender addresses are easily, and commonly, forged.

spam = <*username*>
This condition is available only when Exim is compiled with the content-scanning extension. It causes the incoming message to be scanned by SpamAssassin. For details, see chapter 41.

verify = certificate
This condition is true in an SMTP session if the session is encrypted, and a certificate was received from the client, and the certificate was verified. The server requests a certificate only if the client matches **tls_verify_hosts** or **tls_try_verify_hosts** (see chapter 39).

verify = csa
This condition checks whether the sending host (the client) is authorized to send email. Details of how this works are given in section 40.46.

verify = header_sender/<*options*>
This condition is relevant only in an ACL that is run after a message has been received, that is, in an ACL specified by **acl_smtp_data** or **acl_not_smtp**. It checks that there is a verifiable address in at least one of the *Sender:*, *Reply-To:*, or *From:* header lines. Such an address is loosely thought of as a "sender" address (hence the name of the test). However, an address that appears in one of these headers need not be an address that accepts bounce messages; only sender addresses in envelopes are required to accept bounces. Therefore, if you use the callout option on this check, you might want to arrange for a non-empty address in the MAIL command.

Details of address verification and the options are given later, starting at section 40.40 (callouts are described in section 40.41). You can combine this condition with the **senders** condition to restrict it to bounce messages only:

```
deny    senders = :
        message = A valid sender header is required for bounces
       !verify  = header_sender
```

verify = header_syntax
This condition is relevant only in an ACL that is run after a message has been received, that is, in an ACL specified by **acl_smtp_data** or **acl_not_smtp**. It checks the syntax of all header lines

that can contain lists of addresses (*Sender:*, *From:*, *Reply-To:*, *To:*, *Cc:*, and *Bcc:*). Unqualified addresses (local parts without domains) are permitted only in locally generated messages and from hosts that match **sender_unqualified_hosts** or **recipient_unqualified_hosts**, as appropriate.

Note that this condition is a syntax check only. However, a common spamming ploy used to be to send syntactically invalid headers such as

```
To: @
```

and this condition can be used to reject such messages, though they are not as common as they used to be.

verify = helo
This condition is true if a HELO or EHLO command has been received from the client host, and its contents have been verified. If there has been no previous attempt to verify the HELO/EHLO contents, it is carried out when this condition is encountered. See the description of the **helo_verify_hosts** and **helo_try_verify_hosts** options for details of how to request verification independently of this condition.

For SMTP input that does not come over TCP/IP (the **-bs** command line option), this condition is always true.

verify = not_blind
This condition checks that there are no blind (bcc) recipients in the message. Every envelope recipient must appear either in a *To:* header line or in a *Cc:* header line for this condition to be true. Local parts are checked case-sensitively; domains are checked case-insensitively. If *Resent-To:* or *Resent-Cc:* header lines exist, they are also checked. This condition can be used only in a DATA or non-SMTP ACL.

There are, of course, many legitimate messages that make use of blind (bcc) recipients. This check should not be used on its own for blocking messages.

verify = recipient/<*options*>
This condition is relevant only after a RCPT command. It verifies the current recipient. Details of address verification are given later, starting at section 40.40. After a recipient has been verified, the value of *$address_data* is the last value that was set while routing the address. This applies even if the verification fails. When an address that is being verified is redirected to a single address, verification continues with the new address, and in that case, the subsequent value of *$address_data* is the value for the child address.

verify = reverse_host_lookup
This condition ensures that a verified host name has been looked up from the IP address of the client host. (This may have happened already if the host name was needed for checking a host list, or if the host matched **host_lookup**.) Verification ensures that the host name obtained from a reverse DNS lookup, or one of its aliases, does, when it is itself looked up in the DNS, yield the original IP address.

If this condition is used for a locally generated message (that is, when there is no client host involved), it always succeeds.

verify = sender/<*options*>
This condition is relevant only after a MAIL or RCPT command, or after a message has been received (the **acl_smtp_data** or **acl_not_smtp** ACLs). If the message's sender is empty (that is, this is a bounce message), the condition is true. Otherwise, the sender address is verified.

If there is data in the *$address_data* variable at the end of routing, its value is placed in *$sender_address_data* at the end of verification. This value can be used in subsequent conditions and modifiers in the same ACL statement. It does not persist after the end of the current statement. If you want to preserve the value for longer, you can save it in an ACL variable.

Details of verification are given later, starting at section 40.40. Exim caches the result of sender verification, to avoid doing it more than once per message.

verify = sender=<*address*>**/**<*options*>
This is a variation of the previous option, in which a modified address is verified as a sender.

40.24 Using DNS lists

In its simplest form, the **dnslists** condition tests whether the calling host is on at least one of a number of DNS lists by looking up the inverted IP address in one or more DNS domains. For example, if the calling host's IP address is 192.168.62.43, and the ACL statement is

```
deny dnslists = blackholes.mail-abuse.org : \
                dialups.mail-abuse.org
```

the following records are looked up:

```
43.62.168.192.blackholes.mail-abuse.org
43.62.168.192.dialups.mail-abuse.org
```

As soon as Exim finds an existing DNS record, processing of the list stops. Thus, multiple entries on the list provide an "or" conjunction. If you want to test that a host is on more than one list (an "and" conjunction), you can use two separate conditions:

```
deny dnslists = blackholes.mail-abuse.org
     dnslists = dialups.mail-abuse.org
```

If a DNS lookup times out or otherwise fails to give a decisive answer, Exim behaves as if the host does not match the list item, that is, as if the DNS record does not exist. If there are further items in the DNS list, they are processed.

This is usually the required action when **dnslists** is used with **deny** (which is the most common usage), because it prevents a DNS failure from blocking mail. However, you can change this behaviour by putting one of the following special items in the list:

`+include_unknown`	behave as if the item is on the list
`+exclude_unknown`	behave as if the item is not on the list (default)
`+defer_unknown`	give a temporary error

Each of these applies to any subsequent items on the list. For example:

```
deny dnslists = +defer_unknown : foo.bar.example
```

Testing the list of domains stops as soon as a match is found. If you want to warn for one list and block for another, you can use two different statements:

```
deny  dnslists = blackholes.mail-abuse.org
warn  message  = X-Warn: sending host is on dialups list
      dnslists = dialups.mail-abuse.org
```

DNS list lookups are cached by Exim for the duration of the SMTP session, so a lookup based on the IP address is done at most once for any incoming connection. Exim does not share information between multiple incoming connections (but your local name server cache should be active).

40.25 Specifying the IP address for a DNS list lookup

By default, the IP address that is used in a DNS list lookup is the IP address of the calling host. However, you can specify another IP address by listing it after the domain name, introduced by a slash. For example:

```
deny dnslists = black.list.tld/192.168.1.2
```

This feature is not very helpful with explicit IP addresses; it is intended for use with IP addresses that are looked up, for example, the IP addresses of the MX hosts or nameservers of an email sender address. For an example, see section 40.27 below.

40.26 DNS lists keyed on domain names

There are some lists that are keyed on domain names rather than inverted IP addresses (see for example the *domain based zones* link at **http://www.rfc-ignorant.org/**). No reversing of components is used with these lists. You can change the name that is looked up in a DNS list by listing it after the domain name, introduced by a slash. For example,

```
deny  message  = Sender's domain is listed at $dnslist_domain
      dnslists = dsn.rfc-ignorant.org/$sender_address_domain
```

This particular example is useful only in ACLs that are obeyed after the RCPT or DATA commands, when a sender address is available. If (for example) the message's sender is *user@tld.example* the name that is looked up by this example is

```
tld.example.dsn.rfc-ignorant.org
```

A single **dnslists** condition can contain entries for both names and IP addresses. For example:

```
deny dnslists = sbl.spamhaus.org : \
                dsn.rfc-ignorant.org/$sender_address_domain
```

The first item checks the sending host's IP address; the second checks a domain name. The whole condition is true if either of the DNS lookups succeeds.

40.27 Multiple explicit keys for a DNS list

The syntax described above for looking up explicitly-defined values (either names or IP addresses) in a DNS blacklist is a simplification. After the domain name for the DNS list, what follows the slash can in fact be a list of items. As with all lists in Exim, the default separator is a colon. However, because this is a sublist within the list of DNS blacklist domains, it is necessary either to double the separators like this:

```
dnslists = black.list.tld/name.1::name.2
```

or to change the separator character, like this:

```
dnslists = black.list.tld/<;name.1;name.2
```

If an item in the list is an IP address, it is inverted before the DNS blacklist domain is appended. If it is not an IP address, no inversion occurs. Consider this condition:

```
dnslists = black.list.tld/<;192.168.1.2;a.domain
```

The DNS lookups that occur are:

```
2.1.168.192.black.list.tld
a.domain.black.list.tld
```

Once a DNS record has been found (that matches a specific IP return address, if specified – see section 40.30), no further lookups are done. If there is a temporary DNS error, the rest of the sublist of domains or IP addresses is tried. A temporary error for the whole dnslists item occurs only if no other DNS lookup in this sublist succeeds. In other words, a successful lookup for any of the items in the sublist overrides a temporary error for a previous item.

The ability to supply a list of items after the slash is in some sense just a syntactic convenience. These two examples have the same effect:

```
dnslists = black.list.tld/a.domain : black.list.tld/b.domain
dnslists = black.list.tld/a.domain::b.domain
```

However, when the data for the list is obtained from a lookup, the second form is usually much more convenient. Consider this example:

```
deny message  = The mail servers for the domain \
                $sender_address_domain \
                are listed at $dnslist_domain ($dnslist_value); \
                see $dnslist_text.
     dnslists = sbl.spamhaus.org/<|${lookup dnsdb {>|a=<|\
                                  ${lookup dnsdb {>|mxh=\
                                  $sender_address_domain} }} }
```

Note the use of >| in the dnsdb lookup to specify the separator for multiple DNS records. The inner dnsdb lookup produces a list of MX hosts and the outer dnsdb lookup finds the IP addresses for these hosts. The result of expanding the condition might be something like this:

```
dnslists = sbl.spahmaus.org/<|192.168.2.3|192.168.5.6|...
```

Thus, this example checks whether or not the IP addresses of the sender domain's mail servers are on the Spamhaus black list.

The key that was used for a successful DNS list lookup is put into the variable *$dnslist_matched* (see section 40.29).

40.28 Data returned by DNS lists

DNS lists are constructed using address records in the DNS. The original RBL just used the address 127.0.0.1 on the right hand side of each record, but the RBL+ list and some other lists use a number of values with different meanings. The values used on the RBL+ list are:

127.1.0.1 RBL
127.1.0.2 DUL
127.1.0.3 DUL and RBL
127.1.0.4 RSS
127.1.0.5 RSS and RBL
127.1.0.6 RSS and DUL
127.1.0.7 RSS and DUL and RBL

Section 40.30 below describes how you can distinguish between different values. Some DNS lists may return more than one address record; see section 40.32 for details of how they are checked.

40.29 Variables set from DNS lists

When an entry is found in a DNS list, the variable *$dnslist_domain* contains the name of the overall domain that matched (for example, `spamhaus.example`), *$dnslist_matched* contains the key within that domain (for example, `192.168.5.3`), and *$dnslist_value* contains the data from the DNS record. When the key is an IP address, it is not reversed in *$dnslist_matched* (though it is, of course, in the actual lookup). In simple cases, for example:

```
deny dnslists = spamhaus.example
```

the key is also available in another variable (in this case, *$sender_host_address*). In more complicated cases, however, this is not true. For example, using a data lookup (as described in section 40.27) might generate a dnslists lookup like this:

```
deny dnslists = spamhaus.example/<|192.168.1.2|192.168.6.7|...
```

If this condition succeeds, the value in *$dnslist_matched* might be `192.168.6.7` (for example).

If more than one address record is returned by the DNS lookup, all the IP addresses are included in *$dnslist_value*, separated by commas and spaces. The variable *$dnslist_text* contains the contents of any associated TXT record. For lists such as RBL+ the TXT record for a merged entry is often not very meaningful. See section 40.33 for a way of obtaining more information.

You can use the DNS list variables in **message** or **log_message** modifiers – although these appear before the condition in the ACL, they are not expanded until after it has failed. For example:

```
deny    hosts = !+local_networks
        message = $sender_host_address is listed \
                  at $dnslist_domain
        dnslists = rbl-plus.mail-abuse.example
```

40.30 Additional matching conditions for DNS lists

You can add an equals sign and an IP address after a **dnslists** domain name in order to restrict its action to DNS records with a matching right hand side. For example,

```
deny dnslists = rblplus.mail-abuse.org=127.0.0.2
```

rejects only those hosts that yield 127.0.0.2. Without this additional data, any address record is considered to be a match. For the moment, we assume that the DNS lookup returns just one record. Section 40.32 describes how multiple records are handled.

More than one IP address may be given for checking, using a comma as a separator. These are alternatives – if any one of them matches, the **dnslists** condition is true. For example:

```
deny  dnslists = a.b.c=127.0.0.2,127.0.0.3
```

If you want to specify a constraining address list and also specify names or IP addresses to be looked up, the constraining address list must be specified first. For example:

```
deny dnslists = dsn.rfc-ignorant.org\
                =127.0.0.2/$sender_address_domain
```

If the character `&` is used instead of =, the comparison for each listed IP address is done by a bitwise "and" instead of by an equality test. In other words, the listed addresses are used as bit masks. The comparison is true if all the bits in the mask are present in the address that is being tested. For example:

```
dnslists = a.b.c&0.0.0.3
```

matches if the address is *x.x.x.*3, *x.x.x.*7, *x.x.x.*11, etc. If you want to test whether one bit or another bit is present (as opposed to both being present), you must use multiple values. For example:

```
dnslists = a.b.c&0.0.0.1,0.0.0.2
```

matches if the final component of the address is an odd number or two times an odd number.

40.31 Negated DNS matching conditions

You can supply a negative list of IP addresses as part of a **dnslists** condition. Whereas

```
deny  dnslists = a.b.c=127.0.0.2,127.0.0.3
```

means "deny if the host is in the black list at the domain *a.b.c* and the IP address yielded by the list is either 127.0.0.2 or 127.0.0.3",

```
deny  dnslists = a.b.c!=127.0.0.2,127.0.0.3
```

means "deny if the host is in the black list at the domain *a.b.c* and the IP address yielded by the list is not 127.0.0.2 and not 127.0.0.3". In other words, the result of the test is inverted if an exclamation mark appears before the = (or the `&`) sign.

Note: This kind of negation is not the same as negation in a domain, host, or address list (which is why the syntax is different).

If you are using just one list, the negation syntax does not gain you much. The previous example is precisely equivalent to

```
deny  dnslists = a.b.c
     !dnslists = a.b.c=127.0.0.2,127.0.0.3
```

However, if you are using multiple lists, the negation syntax is clearer. Consider this example:

```
deny  dnslists = sbl.spamhaus.org : \
                 list.dsbl.org : \
                 dnsbl.njabl.org!=127.0.0.3 : \
                 relays.ordb.org
```

Using only positive lists, this would have to be:

```
deny  dnslists = sbl.spamhaus.org : \
                 list.dsbl.org
deny  dnslists = dnsbl.njabl.org
     !dnslists = dnsbl.njabl.org=127.0.0.3
deny  dnslists = relays.ordb.org
```

which is less clear, and harder to maintain.

40.32 Handling multiple DNS records from a DNS list

A DNS lookup for a **dnslists** condition may return more than one DNS record, thereby providing more than one IP address. When an item in a **dnslists** list is followed by = or & and a list of IP addresses, in order to restrict the match to specific results from the DNS lookup, there are two ways in which the checking can be handled. For example, consider the condition:

```
dnslists = a.b.c=127.0.0.1
```

What happens if the DNS lookup for the incoming IP address yields both 127.0.0.1 and 127.0.0.2 by means of two separate DNS records? Is the condition true because at least one given value was found, or is it false because at least one of the found values was not listed? And how does this affect negated conditions? Both possibilities are provided for with the help of additional separators == and =&.

- If = or & is used, the condition is true if any one of the looked up IP addresses matches one of the listed addresses. For the example above, the condition is true because 127.0.0.1 matches.
- If == or =& is used, the condition is true only if every one of the looked up IP addresses matches one of the listed addresses. If the condition is changed to:

  ```
  dnslists = a.b.c==127.0.0.1
  ```

  and the DNS lookup yields both 127.0.0.1 and 127.0.0.2, the condition is false because 127.0.0.2 is not listed. You would need to have:

  ```
  dnslists = a.b.c==127.0.0.1,127.0.0.2
  ```

  for the condition to be true.

When ! is used to negate IP address matching, it inverts the result, giving the precise opposite of the behaviour above. Thus:

- If != or !& is used, the condition is true if none of the looked up IP addresses matches one of the listed addresses. Consider:

  ```
  dnslists = a.b.c!&0.0.0.1
  ```

  If the DNS lookup yields both 127.0.0.1 and 127.0.0.2, the condition is false because 127.0.0.1 matches.
- If !== or !=& is used, the condition is true there is at least one looked up IP address that does not match. Consider:

  ```
  dnslists = a.b.c!=&0.0.0.1
  ```

  If the DNS lookup yields both 127.0.0.1 and 127.0.0.2, the condition is true, because 127.0.0.2 does not match. You would need to have:

  ```
  dnslists = a.b.c!=&0.0.0.1,0.0.0.2
  ```

  for the condition to be false.

When the DNS lookup yields only a single IP address, there is no difference between = and == and between & and =&.

40.33 Detailed information from merged DNS lists

When the facility for restricting the matching IP values in a DNS list is used, the text from the TXT record that is set in *$dnslist_text* may not reflect the true reason for rejection. This happens when lists are merged and the IP address in the A record is used to distinguish them; unfortunately there is only one TXT record. One way round this is not to use merged lists, but that can be inefficient because it requires multiple DNS lookups where one would do in the vast majority of cases when the host of interest is not on any of the lists.

A less inefficient way of solving this problem is available. If two domain names, comma-separated, are given, the second is used first to do an initial check, making use of any IP value restrictions that are set. If there is a match, the first domain is used, without any IP value restrictions, to get the TXT record. As a byproduct of this, there is also a check that the IP being tested is indeed on the first list. The first domain is the one that is put in *$dnslist_domain*. For example:

```
reject message  = \
         rejected because $sender_host_address is blacklisted \
         at $dnslist_domain\n$dnslist_text
       dnslists = \
         sbl.spamhaus.org,sbl-xbl.spamhaus.org=127.0.0.2 : \
         dul.dnsbl.sorbs.net,dnsbl.sorbs.net=127.0.0.10
```

For the first blacklist item, this starts by doing a lookup in *sbl-xbl.spamhaus.org* and testing for a 127.0.0.2 return. If there is a match, it then looks in *sbl.spamhaus.org*, without checking the return value, and as long as something is found, it looks for the corresponding TXT record. If there is no match in *sbl-xbl.spamhaus.org*, nothing more is done. The second blacklist item is processed similarly.

If you are interested in more than one merged list, the same list must be given several times, but because the results of the DNS lookups are cached, the DNS calls themselves are not repeated. For example:

```
reject dnslists = \
         http.dnsbl.sorbs.net,dnsbl.sorbs.net=127.0.0.2 : \
         socks.dnsbl.sorbs.net,dnsbl.sorbs.net=127.0.0.3 : \
         misc.dnsbl.sorbs.net,dnsbl.sorbs.net=127.0.0.4 : \
         dul.dnsbl.sorbs.net,dnsbl.sorbs.net=127.0.0.10
```

In this case there is one lookup in *dnsbl.sorbs.net*, and if none of the IP values matches (or if no record is found), this is the only lookup that is done. Only if there is a match is one of the more specific lists consulted.

40.34 DNS lists and IPv6

If Exim is asked to do a dnslist lookup for an IPv6 address, it inverts it nibble by nibble. For example, if the calling host's IP address is 3ffe:ffff:836f:0a00:000a:0800:200a:c031, Exim might look up

```
1.3.0.c.a.0.0.2.0.0.8.0.a.0.0.0.0.0.a.0.f.6.3.8.
  f.f.f.f.e.f.f.3.blackholes.mail-abuse.org
```

(split over two lines here to fit on the page). Unfortunately, some of the DNS lists contain wildcard records, intended for IPv4, that interact badly with IPv6. For example, the DNS entry

```
*.3.some.list.example.    A    127.0.0.1
```

is probably intended to put the entire 3.0.0.0/8 IPv4 network on the list. Unfortunately, it also matches the entire 3::/4 IPv6 network.

You can exclude IPv6 addresses from DNS lookups by making use of a suitable **condition** condition, as in this example:

```
deny   condition = ${if isip4{$sender_host_address}}
       dnslists  = some.list.example
```

40.35 Rate limiting incoming messages

The **ratelimit** ACL condition can be used to measure and control the rate at which clients can send email. This is more powerful than the **smtp_ratelimit_*** options, because those options control the rate of commands in a single SMTP session only, whereas the **ratelimit** condition works across all connections (concurrent and sequential) from the same client host. The syntax of the **ratelimit** condition is:

```
ratelimit = <m> / <p> / <options> / <key>
```

If the average client sending rate is less than *m* messages per time period *p* then the condition is false; otherwise it is true.

As a side-effect, the **ratelimit** condition sets the expansion variable *$sender_rate* to the client's computed rate, *$sender_rate_limit* to the configured value of *m*, and *$sender_rate_period* to the configured value of *p*.

The parameter *p* is the smoothing time constant, in the form of an Exim time interval, for example, `8h` for eight hours. A larger time constant means that it takes Exim longer to forget a client's past behaviour. The parameter *m* is the maximum number of messages that a client is permitted to send in each time interval. It also specifies the number of messages permitted in a fast burst. By increasing both *m* and *p* but keeping *m/p* constant, you can allow a client to send more messages in a burst without changing its overall sending rate limit. Conversely, if *m* and *p* are both small, messages must be sent at an even rate.

There is a script in *util/ratelimit.pl* which extracts sending rates from log files, to assist with choosing appropriate settings for *m* and *p* when deploying the **ratelimit** ACL condition. The script prints usage instructions when it is run with no arguments.

The key is used to look up the data for calculating the client's average sending rate. This data is stored in a database maintained by Exim in its spool directory, alongside the retry and other hints databases. The default key is *$sender_host_address*, which applies the limit to each client host IP address. By changing the key you can change how Exim identifies clients for the purpose of ratelimiting. For example, to limit the sending rate of each authenticated user, independent of the computer they are sending from, set the key to *$authenticated_id*. You must ensure that the lookup key is meaningful; for example, *$authenticated_id* is only meaningful if the client has authenticated, and you can check with the **authenticated** ACL condition.

If you want to limit the rate at which a recipient receives messages, you can use the key `$local_part@$domain` with the **per_rcpt** option (see below) in a RCPT ACL.

Internally, Exim includes the smoothing constant *p* and the options in the lookup key because they alter the meaning of the stored data. This is not true for the limit *m*, so you can alter the configured maximum rate and Exim will still remember clients' past behaviour, but if you alter the other ratelimit parameters Exim forgets past behaviour.

Each **ratelimit** condition can have up to three options. One option specifies what Exim measures the rate of, and the second specifies how Exim handles excessively fast clients. The third option can be `noupdate`, to disable updating of the ratelimiting database (see section 40.39). The options are separated by a slash, like the other parameters. They may appear in any order.

40.36 Ratelimit options for what is being measured

The **per_conn** option limits the client's connection rate.

The **per_mail** option limits the client's rate of sending messages. This is the default if none of the **per_*** options is specified.

The **per_byte** option limits the sender's email bandwidth. Note that it is best to use this option in the DATA ACL; if it is used in an earlier ACL it relies on the SIZE parameter on the MAIL command, which may be inaccurate or completely missing. You can follow the limit *m* in the configuration with K, M, or G to specify limits in kilobytes, megabytes, or gigabytes, respectively.

The **per_cmd** option causes Exim to recompute the rate every time the condition is processed. This can be used to limit the SMTP command rate. The alias **per_rcpt** is provided for use in the RCPT ACL instead of **per_cmd** to make it clear that the effect is to limit the rate at which recipients are accepted. Note that in this case the rate limiting engine will see a message with many recipients as a large high-speed burst.

40.37 Ratelimit options for handling fast clients

If a client's average rate is greater than the maximum, the rate limiting engine can react in two possible ways, depending on the presence of the **strict** or **leaky** options. This is independent of the

other counter-measures (such as rejecting the message) that may be specified by the rest of the ACL. The default mode is leaky, which avoids a sender's over-aggressive retry rate preventing it from getting any email through.

The **strict** option means that the client's recorded rate is always updated. The effect of this is that Exim measures the client's average rate of attempts to send email, which can be much higher than the maximum it is actually allowed. If the client is over the limit it may be subjected to counter-measures in the ACL until it slows down below the maximum rate. The smoothing period determines the time it takes for a high sending rate to decay exponentially to 37% of its peak value, which means that you can work out the time (the number of smoothing periods) that a client is subjected to counter-measures after an over-limit burst with this formula:

```
ln(peakrate/maxrate)
```

The **leaky** (default) option means that the client's recorded rate is not updated if it is above the limit. The effect of this is that Exim measures the client's average rate of successfully sent email, which cannot be greater than the maximum allowed. If the client is over the limit it may suffer some counter-measures (as specified in the ACL), but it will still be able to send email at the configured maximum rate, whatever the rate of its attempts. This is generally the better choice if you have clients that retry automatically.

40.38 Using rate limiting

Exim's other ACL facilities are used to define what counter-measures are taken when the rate limit is exceeded. This might be anything from logging a warning (for example, while measuring existing sending rates in order to define policy), through time delays to slow down fast senders, up to rejecting the message. For example:

```
# Log all senders' rates
warn ratelimit = 0 / 1h / strict
     log_message = Sender rate $sender_rate / $sender_rate_period

# Slow down fast senders; note the need to truncate $sender_rate
# at the decimal point.
warn ratelimit = 100 / 1h / per_rcpt / strict
     delay     = ${eval: ${sg{$sender_rate}{[.].*}{}} - \
                    $sender_rate_limit }s

# Keep authenticated users under control
deny authenticated = *
     ratelimit = 100 / 1d / strict / $authenticated_id

# System-wide rate limit
defer message = Sorry, too busy. Try again later.
     ratelimit = 10 / 1s / $primary_hostname

# Restrict incoming rate from each host, with a default
# set using a macro and special cases looked up in a table.
defer message = Sender rate exceeds $sender_rate_limit \
                messages per $sender_rate_period
     ratelimit = ${lookup {$sender_host_address} \
                    cdb {DB/ratelimits.cdb} \
                    {$value} {RATELIMIT} }
```

Warning: If you have a busy server with a lot of **ratelimit** tests, especially with the **per_rcpt** option, you may suffer from a performance bottleneck caused by locking on the ratelimit hints database. Apart from making your ACLs less complicated, you can reduce the problem by using a RAM disk for Exim's hints directory (usually */var/spool/exim/db/*). However this means that Exim will lose its hints data after a reboot (including retry hints, the callout cache, and ratelimit data).

40.39 Reading ratelimit data without updating

If the **noupdate** option is present on a **ratelimit** ACL condition, Exim computes the rate and checks the limit as normal, but it does not update the saved data. This means that, in relevant ACLs, it is possible to lookup the existence of a specified (or auto-generated) ratelimit key without incrementing the ratelimit counter for that key. In order for this to be useful, another ACL entry must set the rate for the same key (otherwise it will always be zero). For example:

```
acl_check_connect:
  deny ratelimit = 100 / 5m / strict / noupdate
       log_message = RATE: $sender_rate/$sender_rate_period \
                     (max $sender_rate_limit)
```

... some other logic and tests...

```
acl_check_mail:
  warn ratelimit = 100 / 5m / strict / per_cmd
       condition = ${if le{$sender_rate}{$sender_rate_limit}}
       logwrite  = RATE UPDATE: $sender_rate/$sender_rate_period \
                     (max $sender_rate_limit)
```

In this example, the rate is tested and used to deny access (when it is too high) in the connect ACL, but the actual computation of the remembered rate happens later, on a per-command basis, in another ACL.

40.40 Address verification

Several of the **verify** conditions described in section 40.23 cause addresses to be verified. Section 40.44 discusses the reporting of sender verification failures. The verification conditions can be followed by options that modify the verification process. The options are separated from the keyword and from each other by slashes, and some of them contain parameters. For example:

```
verify = sender/callout
verify = recipient/defer_ok/callout=10s,defer_ok
```

The first stage of address verification, which always happens, is to run the address through the routers, in "verify mode". Routers can detect the difference between verification and routing for delivery, and their actions can be varied by a number of generic options such as **verify** and **verify_only** (see chapter 15). If routing fails, verification fails. The available options are as follows:

- If the **callout** option is specified, successful routing to one or more remote hosts is followed by a "callout" to those hosts as an additional check. Callouts and their sub-options are discussed in the next section.
- If there is a defer error while doing verification routing, the ACL normally returns "defer". However, if you include **defer_ok** in the options, the condition is forced to be true instead. Note that this is a main verification option as well as a suboption for callouts.
- The **no_details** option is covered in section 40.44, which discusses the reporting of sender address verification failures.
- The **success_on_redirect** option causes verification always to succeed immediately after a successful redirection. By default, if a redirection generates just one address, that address is also verified. See further discussion in section 40.45.

After an address verification failure, *$acl_verify_message* contains the error message that is associated with the failure. It can be preserved by coding like this:

```
warn  !verify = sender
       set acl_m0 = $acl_verify_message
```

If you are writing your own custom rejection message or log message when denying access, you can use this variable to include information about the verification failure.

In addition, *$sender_verify_failure* or *$recipient_verify_failure* (as appropriate) contains one of the following words:

- **qualify**: The address was unqualified (no domain), and the message was neither local nor came from an exempted host.
- **route**: Routing failed.
- **mail**: Routing succeeded, and a callout was attempted; rejection occurred at or before the MAIL command (that is, on initial connection, HELO, or MAIL).
- **recipient**: The RCPT command in a callout was rejected.
- **postmaster**: The postmaster check in a callout was rejected.

The main use of these variables is expected to be to distinguish between rejections of MAIL and rejections of RCPT in callouts.

40.41 Callout verification

For non-local addresses, routing verifies the domain, but is unable to do any checking of the local part. There are situations where some means of verifying the local part is desirable. One way this can be done is to make an SMTP *callback* to a delivery host for the sender address or a *callforward* to a subsequent host for a recipient address, to see if the host accepts the address. We use the term *callout* to cover both cases. Note that for a sender address, the callback is not to the client host that is trying to deliver the message, but to one of the hosts that accepts incoming mail for the sender's domain.

Exim does not do callouts by default. If you want them to happen, you must request them by setting appropriate options on the **verify** condition, as described below. This facility should be used with care, because it can add a lot of resource usage to the cost of verifying an address. However, Exim does cache the results of callouts, which helps to reduce the cost. Details of caching are in section 40.43.

Recipient callouts are usually used only between hosts that are controlled by the same administration. For example, a corporate gateway host could use callouts to check for valid recipients on an internal mailserver. A successful callout does not guarantee that a real delivery to the address would succeed; on the other hand, a failing callout does guarantee that a delivery would fail.

If the **callout** option is present on a condition that verifies an address, a second stage of verification occurs if the address is successfully routed to one or more remote hosts. The usual case is routing by a *dnslookup* or a *manualroute* router, where the router specifies the hosts. However, if a router that does not set up hosts routes to an *smtp* transport with a **hosts** setting, the transport's hosts are used. If an *smtp* transport has **hosts_override** set, its hosts are always used, whether or not the router supplies a host list.

The port that is used is taken from the transport, if it is specified and is a remote transport. (For routers that do verification only, no transport need be specified.) Otherwise, the default SMTP port is used. If a remote transport specifies an outgoing interface, this is used; otherwise the interface is not specified. Likewise, the text that is used for the HELO command is taken from the transport's **helo_data** option; if there is no transport, the value of *$smtp_active_hostname* is used.

For a sender callout check, Exim makes SMTP connections to the remote hosts, to test whether a bounce message could be delivered to the sender address. The following SMTP commands are sent:

```
HELO <local host name>
MAIL FROM:<>
RCPT TO:<the address to be tested>
QUIT
```

LHLO is used instead of HELO if the transport's **protocol** option is set to "lmtp".

A recipient callout check is similar. By default, it also uses an empty address for the sender. This default is chosen because most hosts do not make use of the sender address when verifying a recipient. Using the same address means that a single cache entry can be used for each recipient. Some

sites, however, do make use of the sender address when verifying. These are catered for by the **use_sender** and **use_postmaster** options, described in the next section.

If the response to the RCPT command is a 2*xx* code, the verification succeeds. If it is 5*xx*, the verification fails. For any other condition, Exim tries the next host, if any. If there is a problem with all the remote hosts, the ACL yields "defer", unless the **defer_ok** parameter of the **callout** option is given, in which case the condition is forced to succeed.

A callout may take a little time. For this reason, Exim normally flushes SMTP output before performing a callout in an ACL, to avoid unexpected timeouts in clients when the SMTP PIPELINING extension is in use. The flushing can be disabled by using a **control** modifier to set **no_callout_flush**.

40.42 Additional parameters for callouts

The **callout** option can be followed by an equals sign and a number of optional parameters, separated by commas. For example:

```
verify = recipient/callout=10s,defer_ok
```

The old syntax, which had **callout_defer_ok** and **check_postmaster** as separate verify options, is retained for backwards compatibility, but is now deprecated. The additional parameters for **callout** are as follows:

<*a time interval*>
: This specifies the timeout that applies for the callout attempt to each host. For example:

  ```
  verify = sender/callout=5s
  ```

  The default is 30 seconds. The timeout is used for each response from the remote host. It is also used for the initial connection, unless overridden by the **connect** parameter.

connect = <*time interval*>
: This parameter makes it possible to set a different (usually smaller) timeout for making the SMTP connection. For example:

  ```
  verify = sender/callout=5s,connect=1s
  ```

  If not specified, this timeout defaults to the general timeout value.

defer_ok
: When this parameter is present, failure to contact any host, or any other kind of temporary error, is treated as success by the ACL. However, the cache is not updated in this circumstance.

fullpostmaster
: This operates like the **postmaster** option (see below), but if the check for *postmaster@domain* fails, it tries just *postmaster*, without a domain, in accordance with the specification in RFC 2821. The RFC states that the unqualified address *postmaster* should be accepted.

mailfrom = <*email address*>
: When verifying addresses in header lines using the **header_sender** verification option, Exim behaves by default as if the addresses are envelope sender addresses from a message. Callout verification therefore tests to see whether a bounce message could be delivered, by using an empty address in the MAIL command. However, it is arguable that these addresses might never be used as envelope senders, and could therefore justifiably reject bounce messages (empty senders). The **mailfrom** callout parameter allows you to specify what address to use in the MAIL command. For example:

  ```
  require  verify = header_sender/callout=mailfrom=abcd@x.y.z
  ```

  This parameter is available only for the **header_sender** verification option.

maxwait = <*time interval*>
: This parameter sets an overall timeout for performing a callout verification. For example:

  ```
  verify = sender/callout=5s,maxwait=30s
  ```

This timeout defaults to four times the callout timeout for individual SMTP commands. The overall timeout applies when there is more than one host that can be tried. The timeout is checked before trying the next host. This prevents very long delays if there are a large number of hosts and all are timing out (for example, when network connections are timing out).

no_cache
When this parameter is given, the callout cache is neither read nor updated.

postmaster
When this parameter is set, a successful callout check is followed by a similar check for the local part *postmaster* at the same domain. If this address is rejected, the callout fails (but see **fullpostmaster** above). The result of the postmaster check is recorded in a cache record; if it is a failure, this is used to fail subsequent callouts for the domain without a connection being made, until the cache record expires.

postmaster_mailfrom = <*email address*>
The postmaster check uses an empty sender in the MAIL command by default. You can use this parameter to do a postmaster check using a different address. For example:

```
require  verify = sender/callout=postmaster_mailfrom=abc@x.y.z
```

If both **postmaster** and **postmaster_mailfrom** are present, the rightmost one overrides. The **postmaster** parameter is equivalent to this example:

```
require  verify = sender/callout=postmaster_mailfrom=
```

Warning: The caching arrangements for postmaster checking do not take account of the sender address. It is assumed that either the empty address or a fixed non-empty address will be used. All that Exim remembers is that the postmaster check for the domain succeeded or failed.

random
When this parameter is set, before doing the normal callout check, Exim does a check for a "random" local part at the same domain. The local part is not really random – it is defined by the expansion of the option **callout_random_local_part**, which defaults to

```
$primary_host_name-$tod_epoch-testing
```

The idea here is to try to determine whether the remote host accepts all local parts without checking. If it does, there is no point in doing callouts for specific local parts. If the "random" check succeeds, the result is saved in a cache record, and used to force the current and subsequent callout checks to succeed without a connection being made, until the cache record expires.

use_postmaster
This parameter applies to recipient callouts only. For example:

```
deny  !verify = recipient/callout=use_postmaster
```

It causes a non-empty postmaster address to be used in the MAIL command when performing the callout for the recipient, and also for a "random" check if that is configured. The local part of the address is `postmaster` and the domain is the contents of *$qualify_domain*.

use_sender
This option applies to recipient callouts only. For example:

```
require  verify = recipient/callout=use_sender
```

It causes the message's actual sender address to be used in the MAIL command when performing the callout, instead of an empty address. There is no need to use this option unless you know that the called hosts make use of the sender when checking recipients. If used indiscriminately, it reduces the usefulness of callout caching.

If you use any of the parameters that set a non-empty sender for the MAIL command (**mailfrom**, **postmaster_mailfrom**, **use_postmaster**, or **use_sender**), you should think about possible loops. Recipient checking is usually done between two hosts that are under the same management, and the host that receives the callouts is not normally configured to do callouts itself. Therefore, it is normally safe to use **use_postmaster** or **use_sender** in these circumstances.

However, if you use a non-empty sender address for a callout to an arbitrary host, there is the likelihood that the remote host will itself initiate a callout check back to your host. As it is checking what appears to be a message sender, it is likely to use an empty address in MAIL, thus avoiding a callout loop. However, to be on the safe side it would be best to set up your own ACLs so that they do not do sender verification checks when the recipient is the address you use for header sender or postmaster callout checking.

Another issue to think about when using non-empty senders for callouts is caching. When you set **mailfrom** or **use_sender**, the cache record is keyed by the sender/recipient combination; thus, for any given recipient, many more actual callouts are performed than when an empty sender or postmaster is used.

40.43 Callout caching

Exim caches the results of callouts in order to reduce the amount of resources used, unless you specify the **no_cache** parameter with the **callout** option. A hints database called "callout" is used for the cache. Two different record types are used: one records the result of a callout check for a specific address, and the other records information that applies to the entire domain (for example, that it accepts the local part *postmaster*).

When an original callout fails, a detailed SMTP error message is given about the failure. However, for subsequent failures use the cache data, this message is not available.

The expiry times for negative and positive address cache records are independent, and can be set by the global options **callout_negative_expire** (default 2h) and **callout_positive_expire** (default 24h), respectively.

If a host gives a negative response to an SMTP connection, or rejects any commands up to and including

```
MAIL FROM:<>
```

(but not including the MAIL command with a non-empty address), any callout attempt is bound to fail. Exim remembers such failures in a domain cache record, which it uses to fail callouts for the domain without making new connections, until the domain record times out. There are two separate expiry times for domain cache records: **callout_domain_negative_expire** (default 3h) and **callout_domain_positive_expire** (default 7d).

Domain records expire when the negative expiry time is reached if callouts cannot be made for the domain, or if the postmaster check failed. Otherwise, they expire when the positive expiry time is reached. This ensures that, for example, a host that stops accepting "random" local parts will eventually be noticed.

The callout caching mechanism is based on the domain of the address that is being tested. If the domain routes to several hosts, it is assumed that their behaviour will be the same.

40.44 Sender address verification reporting

See section 40.40 for a general discussion of verification. When sender verification fails in an ACL, the details of the failure are given as additional output lines before the 550 response to the relevant SMTP command (RCPT or DATA). For example, if sender callout is in use, you might see:

```
MAIL FROM:<xyz@abc.example>
250 OK
RCPT TO:<pqr@def.example>
550-Verification failed for <xyz@abc.example>
550-Called:   192.168.34.43
550-Sent:     RCPT TO:<xyz@abc.example>
550-Response: 550 Unknown local part xyz in <xyz@abc.example>
550 Sender verification failed
```

If more than one RCPT command fails in the same way, the details are given only for the first of them. However, some administrators do not want to send out this much information. You can suppress

the details by adding `/no_details` to the ACL statement that requests sender verification. For example:

```
verify = sender/no_details
```

40.45 Redirection while verifying

A dilemma arises when a local address is redirected by aliasing or forwarding during verification: should the generated addresses themselves be verified, or should the successful expansion of the original address be enough to verify it? By default, Exim takes the following pragmatic approach:

- When an incoming address is redirected to just one child address, verification continues with the child address, and if that fails to verify, the original verification also fails.
- When an incoming address is redirected to more than one child address, verification does not continue. A success result is returned.

This seems the most reasonable behaviour for the common use of aliasing as a way of redirecting different local parts to the same mailbox. It means, for example, that a pair of alias entries of the form

```
A.Wol:   aw123
aw123:   :fail: Gone away, no forwarding address
```

work as expected, with both local parts causing verification failure. When a redirection generates more than one address, the behaviour is more like a mailing list, where the existence of the alias itself is sufficient for verification to succeed.

It is possible, however, to change the default behaviour so that all successful redirections count as successful verifications, however many new addresses are generated. This is specified by the **success_on_redirect** verification option. For example:

```
require verify = recipient/success_on_redirect/callout=10s
```

In this example, verification succeeds if a router generates a new address, and the callout does not occur, because no address was routed to a remote host.

When verification is being tested via the **-bv** option, the treatment of redirections is as just described, unless the **-v** or any debugging option is also specified. In that case, full verification is done for every generated address and a report is output for each of them.

40.46 Client SMTP authorization (CSA)

Client SMTP Authorization is a system that allows a site to advertise which machines are and are not permitted to send email. This is done by placing special SRV records in the DNS; these are looked up using the client's HELO domain. At the time of writing, CSA is still an Internet Draft. Client SMTP Authorization checks in Exim are performed by the ACL condition:

```
verify = csa
```

This fails if the client is not authorized. If there is a DNS problem, or if no valid CSA SRV record is found, or if the client is authorized, the condition succeeds. These three cases can be distinguished using the expansion variable *$csa_status*, which can take one of the values "fail", "defer", "unknown", or "ok". The condition does not itself defer because that would be likely to cause problems for legitimate email.

The error messages produced by the CSA code include slightly more detail. If *$csa_status* is "defer", this may be because of problems looking up the CSA SRV record, or problems looking up the CSA target address record. There are four reasons for *$csa_status* being "fail":

- The client's host name is explicitly not authorized.
- The client's IP address does not match any of the CSA target IP addresses.
- The client's host name is authorized but it has no valid target IP addresses (for example, the target's addresses are IPv6 and the client is using IPv4).

- The client's host name has no CSA SRV record but a parent domain has asserted that all subdomains must be explicitly authorized.

The **csa** verification condition can take an argument which is the domain to use for the DNS query. The default is:

```
verify = csa/$sender_helo_name
```

This implementation includes an extension to CSA. If the query domain is an address literal such as [192.0.2.95], or if it is a bare IP address, Exim searches for CSA SRV records in the reverse DNS as if the HELO domain was (for example) *95.2.0.192.in-addr.arpa*. Therefore it is meaningful to say:

```
verify = csa/$sender_host_address
```

In fact, this is the check that Exim performs if the client does not say HELO. This extension can be turned off by setting the main configuration option **dns_csa_use_reverse** to be false.

If a CSA SRV record is not found for the domain itself, a search is performed through its parent domains for a record which might be making assertions about subdomains. The maximum depth of this search is limited using the main configuration option **dns_csa_search_limit**, which is 5 by default. Exim does not look for CSA SRV records in a top level domain, so the default settings handle HELO domains as long as seven (*hostname.five.four.three.two.one.com*). This encompasses the vast majority of legitimate HELO domains.

The *dnsdb* lookup also has support for CSA. Although *dnsdb* also supports direct SRV lookups, this is not sufficient because of the extra parent domain search behaviour of CSA, and (as with PTR lookups) *dnsdb* also turns IP addresses into lookups in the reverse DNS space. The result of a successful lookup such as:

```
${lookup dnsdb {csa=$sender_helo_name}}
```

has two space-separated fields: an authorization code and a target host name. The authorization code can be "Y" for yes, "N" for no, "X" for explicit authorization required but absent, or "?" for unknown.

40.47 Bounce address tag validation

Bounce address tag validation (BATV) is a scheme whereby the envelope senders of outgoing messages have a cryptographic, timestamped "tag" added to them. Genuine incoming bounce messages should therefore always be addressed to recipients that have a valid tag. This scheme is a way of detecting unwanted bounce messages caused by sender address forgeries (often called "collateral spam"), because the recipients of such messages do not include valid tags.

There are two expansion items to help with the implementation of the BATV "prvs" (private signature) scheme in an Exim configuration. This scheme signs the original envelope sender address by using a simple key to add a hash of the address and some time-based randomizing information. The **prvs** expansion item creates a signed address, and the **prvscheck** expansion item checks one. The syntax of these expansion items is described in section 11.5.

As an example, suppose the secret per-address keys are stored in an MySQL database. A query to look up the key for an address could be defined as a macro like this:

```
PRVSCHECK_SQL = ${lookup mysql{SELECT secret FROM batv_prvs \
                WHERE sender='${quote_mysql:$prvscheck_address}'\
                }{$value}}
```

Suppose also that the senders who make use of BATV are defined by an address list called **batv_senders**. Then, in the ACL for RCPT commands, you could use this:

```
# Bounces: drop unsigned addresses for BATV senders
deny message = This address does not send an unsigned reverse path
     senders = :
     recipients = +batv_senders

# Bounces: In case of prvs-signed address, check signature.
```

```
deny message = Invalid reverse path signature.
     senders = :
     condition  = ${prvscheck {$local_part@$domain}\
                  {PRVSCHECK_SQL}{1}}
     !condition = $prvscheck_result
```

The first statement rejects recipients for bounce messages that are addressed to plain BATV sender addresses, because it is known that BATV senders do not send out messages with plain sender addresses. The second statement rejects recipients that are prvs-signed, but with invalid signatures (either because the key is wrong, or the signature has timed out).

A non-prvs-signed address is not rejected by the second statement, because the **prvscheck** expansion yields an empty string if its first argument is not a prvs-signed address, thus causing the **condition** condition to be false. If the first argument is a syntactically valid prvs-signed address, the yield is the third string (in this case "1"), whether or not the cryptographic and timeout checks succeed. The *$prvscheck_result* variable contains the result of the checks (empty for failure, "1" for success).

There are two more issues you must consider when implementing prvs-signing. Firstly, you need to ensure that prvs-signed addresses are not blocked by your ACLs. A prvs-signed address contains a slash character, but the default Exim configuration contains this statement in the RCPT ACL:

```
deny    message       = Restricted characters in address
        domains       = +local_domains
        local_parts   = ^[.] : ^.*[@%!/|]
```

This is a conservative rule that blocks local parts that contain slashes. You should remove the slash in the last line.

Secondly, you have to ensure that the routers accept prvs-signed addresses and deliver them correctly. The easiest way to handle this is to use a *redirect* router to remove the signature with a configuration along these lines:

```
batv_redirect:
  driver = redirect
  data = ${prvscheck {$local_part@$domain}{PRVSCHECK_SQL}}
```

This works because, if the third argument of **prvscheck** is empty, the result of the expansion of a prvs-signed address is the decoded value of the original address. This router should probably be the first of your routers that handles local addresses.

To create BATV-signed addresses in the first place, a transport of this form can be used:

```
external_smtp_batv:
  driver = smtp
  return_path = ${prvs {$return_path} \
                       {${lookup mysql{SELECT \
                       secret FROM batv_prvs WHERE \
                       sender='${quote_mysql:$sender_address}'} \
                       {$value}fail}}}
```

If no key can be found for the existing return path, no signing takes place.

40.48 Using an ACL to control relaying

An MTA is said to *relay* a message if it receives it from some host and delivers it directly to another host as a result of a remote address contained within it. Redirecting a local address via an alias or forward file and then passing the message on to another host is not relaying, but a redirection as a result of the "percent hack" is.

Two kinds of relaying exist, which are termed "incoming" and "outgoing". A host which is acting as a gateway or an MX backup is concerned with incoming relaying from arbitrary hosts to a specific set of domains. On the other hand, a host which is acting as a smart host for a number of clients is concerned with outgoing relaying from those clients to the Internet at large. Often the same host is fulfilling both functions, but in principle these two kinds of relaying are entirely independent. What is

not wanted is the transmission of mail from arbitrary remote hosts through your system to arbitrary domains.

You can implement relay control by means of suitable statements in the ACL that runs for each RCPT command. For convenience, it is often easiest to use Exim's named list facility to define the domains and hosts involved. For example, suppose you want to do the following:

- Deliver a number of domains to mailboxes on the local host (or process them locally in some other way). Let's say these are *my.dom1.example* and *my.dom2.example*.
- Relay mail for a number of other domains for which you are the secondary MX. These might be *friend1.example* and *friend2.example*.
- Relay mail from the hosts on your local LAN, to whatever domains are involved. Suppose your LAN is 192.168.45.0/24.

In the main part of the configuration, you put the following definitions:

```
domainlist local_domains = my.dom1.example : my.dom2.example
domainlist relay_domains = friend1.example : friend2.example
hostlist   relay_hosts   = 192.168.45.0/24
```

Now you can use these definitions in the ACL that is run for every RCPT command:

```
acl_check_rcpt:
  accept domains = +local_domains : +relay_domains
  accept hosts   = +relay_hosts
```

The first statement accepts any RCPT command that contains an address in the local or relay domains. For any other domain, control passes to the second statement, which accepts the command only if it comes from one of the relay hosts. In practice, you will probably want to make your ACL more sophisticated than this, for example, by including sender and recipient verification. The default configuration includes a more comprehensive example, which is described in chapter 7.

40.49 Checking a relay configuration

You can check the relay characteristics of your configuration in the same way that you can test any ACL behaviour for an incoming SMTP connection, by using the **-bh** option to run a fake SMTP session with which you interact.

For specifically testing for unwanted relaying, the host *relay-test.mail-abuse.org* provides a useful service. If you telnet to this host from the host on which Exim is running, using the normal telnet port, you will see a normal telnet connection message and then quite a long delay. Be patient. The remote host is making an SMTP connection back to your host, and trying a number of common probes to test for open relay vulnerability. The results of the tests will eventually appear on your terminal.

41. Content scanning at ACL time

The extension of Exim to include content scanning at ACL time, formerly known as "exiscan", was originally implemented as a patch by Tom Kistner. The code was integrated into the main source for Exim release 4.50, and Tom continues to maintain it. Most of the wording of this chapter is taken from Tom's specification.

It is also possible to scan the content of messages at other times. The *local_scan()* function (see chapter 42) allows for content scanning after all the ACLs have run. A transport filter can be used to scan messages at delivery time (see the **transport_filter** option, described in chapter 24).

If you want to include the ACL-time content-scanning features when you compile Exim, you need to arrange for WITH_CONTENT_SCAN to be defined in your *Local/Makefile*. When you do that, the Exim binary is built with:

- Two additional ACLs (**acl_smtp_mime** and **acl_not_smtp_mime**) that are run for all MIME parts for SMTP and non-SMTP messages, respectively.
- Additional ACL conditions and modifiers: **decode**, **malware**, **mime_regex**, **regex**, and **spam**. These can be used in the ACL that is run at the end of message reception (the **acl_smtp_data** ACL).
- An additional control feature ("no_mbox_unspool") that saves spooled copies of messages, or parts of messages, for debugging purposes.
- Additional expansion variables that are set in the new ACL and by the new conditions.
- Two new main configuration options: **av_scanner** and **spamd_address**.

There is another content-scanning configuration option for *Local/Makefile*, called WITH_OLD_DEMIME. If this is set, the old, deprecated **demime** ACL condition is compiled, in addition to all the other content-scanning features.

Content-scanning is continually evolving, and new features are still being added. While such features are still unstable and liable to incompatible changes, they are made available in Exim by setting options whose names begin EXPERIMENTAL_ in *Local/Makefile*. Such features are not documented in this manual. You can find out about them by reading the file called *doc/experimental.txt*.

All the content-scanning facilities work on a MBOX copy of the message that is temporarily created in a file called:

<*spool_directory*>`/scan/`<*message_id*>`/`<*message_id*>`.eml`

The *.eml* extension is a friendly hint to virus scanners that they can expect an MBOX-like structure inside that file. The file is created when the first content scanning facility is called. Subsequent calls to content scanning conditions open the same file again. The directory is recursively removed when the **acl_smtp_data** ACL has finished running, unless

```
control = no_mbox_unspool
```

has been encountered. When the MIME ACL decodes files, they are put into the same directory by default.

41.1 Scanning for viruses

The **malware** ACL condition lets you connect virus scanner software to Exim. It supports a "generic" interface to scanners called via the shell, and specialized interfaces for "daemon" type virus scanners, which are resident in memory and thus are much faster.

You can set the **av_scanner** option in first part of the Exim configuration file to specify which scanner to use, together with any additional options that are needed. The basic syntax is as follows:

```
av_scanner = <scanner-type>:<option1>:<option2>:[...]
```

If you do not set **av_scanner**, it defaults to

```
av_scanner = sophie:/var/run/sophie
```

If the value of **av_scanner** starts with dollar character, it is expanded before use. The following scanner types are supported in this release:

aveserver
This is the scanner daemon of Kaspersky Version 5. You can get a trial version at **http://www.kaspersky.com**. This scanner type takes one option, which is the path to the daemon's UNIX socket. The default is shown in this example:

```
av_scanner = aveserver:/var/run/aveserver
```

clamd
This daemon-type scanner is GPL and free. You can get it at **http://www.clamav.net/**. Some older versions of clamd do not seem to unpack MIME containers, so it used to be recommended to unpack MIME attachments in the MIME ACL. This no longer believed to be necessary. One option is required: either the path and name of a UNIX socket file, or a hostname or IP number, and a port, separated by space, as in the second of these examples:

```
av_scanner = clamd:/opt/clamd/socket
av_scanner = clamd:192.168.2.100 1234
```

If the option is unset, the default is */tmp/clamd*. Thanks to David Saez for contributing the code for this scanner.

cmdline
This is the keyword for the generic command line scanner interface. It can be used to attach virus scanners that are invoked from the shell. This scanner type takes 3 mandatory options:

(1) The full path and name of the scanner binary, with all command line options, and a placeholder (`%s`) for the directory to scan.

(2) A regular expression to match against the STDOUT and STDERR output of the virus scanner. If the expression matches, a virus was found. You must make absolutely sure that this expression matches on "virus found". This is called the "trigger" expression.

(3) Another regular expression, containing exactly one pair of parentheses, to match the name of the virus found in the scanners output. This is called the "name" expression.

For example, Sophos Sweep reports a virus on a line like this:

```
Virus 'W32/Magistr-B' found in file ./those.bat
```

For the trigger expression, we can match the phrase "found in file". For the name expression, we want to extract the W32/Magistr-B string, so we can match for the single quotes left and right of it. Altogether, this makes the configuration setting:

```
av_scanner = cmdline:\
             /path/to/sweep -ss -all -rec -archive %s:\
             found in file:'(.+)'
```

drweb
The DrWeb daemon scanner (**http://www.sald.com/**) interface takes one argument, either a full path to a UNIX socket, or an IP address and port separated by white space, as in these examples:

```
av_scanner = drweb:/var/run/drwebd.sock
av_scanner = drweb:192.168.2.20 31337
```

If you omit the argument, the default path */usr/local/drweb/run/drwebd.sock* is used. Thanks to Alex Miller for contributing the code for this scanner.

fsecure
The F-Secure daemon scanner (**http://www.f-secure.com**) takes one argument which is the path to a UNIX socket. For example:

```
av_scanner = fsecure:/path/to/.fsav
```

If no argument is given, the default is */var/run/.fsav*. Thanks to Johan Thelmen for contributing the code for this scanner.

kavdaemon
This is the scanner daemon of Kaspersky Version 4. This version of the Kaspersky scanner is outdated. Please upgrade (see **aveserver** above). This scanner type takes one option, which is the path to the daemon's UNIX socket. For example:

```
av_scanner = kavdaemon:/opt/AVP/AvpCtl
```

The default path is */var/run/AvpCtl*.

mksd
This is a daemon type scanner that is aimed mainly at Polish users, though some parts of documentation are now available in English. You can get it at **http://linux.mks.com.pl/**. The only option for this scanner type is the maximum number of processes used simultaneously to scan the attachments, provided that the demime facility is employed and also provided that mksd has been run with at least the same number of child processes. For example:

```
av_scanner = mksd:2
```

You can safely omit this option (the default value is 1).

sophie
Sophie is a daemon that uses Sophos' **libsavi** library to scan for viruses. You can get Sophie at **http://www.clanfield.info/sophie/**. The only option for this scanner type is the path to the UNIX socket that Sophie uses for client communication. For example:

```
av_scanner = sophie:/tmp/sophie
```

The default path is */var/run/sophie*, so if you are using this, you can omit the option.

When **av_scanner** is correctly set, you can use the **malware** condition in the DATA ACL. **Note**: You cannot use the **malware** condition in the MIME ACL.

The **av_scanner** option is expanded each time **malware** is called. This makes it possible to use different scanners. See further below for an example. The **malware** condition caches its results, so when you use it multiple times for the same message, the actual scanning process is only carried out once. However, using expandable items in **av_scanner** disables this caching, in which case each use of the **malware** condition causes a new scan of the message.

The **malware** condition takes a right-hand argument that is expanded before use. It can then be one of

- "true", "*", or "1", in which case the message is scanned for viruses. The condition succeeds if a virus was found, and fail otherwise. This is the recommended usage.
- "false" or "0" or an empty string, in which case no scanning is done and the condition fails immediately.
- A regular expression, in which case the message is scanned for viruses. The condition succeeds if a virus is found and its name matches the regular expression. This allows you to take special actions on certain types of virus.

You can append `/defer_ok` to the **malware** condition to accept messages even if there is a problem with the virus scanner. Otherwise, such a problem causes the ACL to defer.

When a virus is found, the condition sets up an expansion variable called *$malware_name* that contains the name of the virus. You can use it in a **message** modifier that specifies the error returned to the sender, and/or in logging data.

If your virus scanner cannot unpack MIME and TNEF containers itself, you should use the **demime** condition (see section 41.6) before the **malware** condition.

Here is a very simple scanning example:

```
deny message = This message contains malware ($malware_name)
     demime = *
     malware = *
```

The next example accepts messages when there is a problem with the scanner:

```
deny message = This message contains malware ($malware_name)
     demime = *
     malware = */defer_ok
```

The next example shows how to use an ACL variable to scan with both sophie and aveserver. It assumes you have set:

```
av_scanner = $acl_m0
```

in the main Exim configuration.

```
deny message = This message contains malware ($malware_name)
     set acl_m0 = sophie
     malware = *

deny message = This message contains malware ($malware_name)
     set acl_m0 = aveserver
     malware = *
```

41.2 Scanning with SpamAssassin

The **spam** ACL condition calls SpamAssassin's **spamd** daemon to get a spam score and a report for the message. You can get SpamAssassin at **http://www.spamassassin.org**, or, if you have a working Perl installation, you can use CPAN by running:

```
perl -MCPAN -e 'install Mail::SpamAssassin'
```

SpamAssassin has its own set of configuration files. Please review its documentation to see how you can tweak it. The default installation should work nicely, however.

After having installed and configured SpamAssassin, start the **spamd** daemon. By default, it listens on 127.0.0.1, TCP port 783. If you use another host or port for **spamd**, you must set the **spamd_address** option in the global part of the Exim configuration as follows (example):

```
spamd_address = 192.168.99.45 387
```

You do not need to set this option if you use the default. As of version 2.60, **spamd** also supports communication over UNIX sockets. If you want to use these, supply **spamd_address** with an absolute file name instead of a address/port pair:

```
spamd_address = /var/run/spamd_socket
```

You can have multiple **spamd** servers to improve scalability. These can reside on other hardware reachable over the network. To specify multiple **spamd** servers, put multiple address/port pairs in the **spamd_address** option, separated with colons:

```
spamd_address = 192.168.2.10 783 : \
                192.168.2.11 783 : \
                192.168.2.12 783
```

Up to 32 **spamd** servers are supported. The servers are queried in a random fashion. When a server fails to respond to the connection attempt, all other servers are tried until one succeeds. If no server responds, the **spam** condition defers.

Warning: It is not possible to use the UNIX socket connection method with multiple **spamd** servers.

41.3 Calling SpamAssassin from an Exim ACL

Here is a simple example of the use of the **spam** condition in a DATA ACL:

```
deny message = This message was classified as SPAM
     spam = joe
```

The right-hand side of the **spam** condition specifies a name. This is relevant if you have set up multiple SpamAssassin profiles. If you do not want to scan using a specific profile, but rather use the SpamAssassin system-wide default profile, you can scan for an unknown name, or simply use "nobody". However, you must put something on the right-hand side.

The name allows you to use per-domain or per-user antispam profiles in principle, but this is not straightforward in practice, because a message may have multiple recipients, not necessarily all in the same domain. Because the **spam** condition has to be called from a DATA ACL in order to be able to read the contents of the message, the variables *$local_part* and *$domain* are not set.

The right-hand side of the **spam** condition is expanded before being used, so you can put lookups or conditions there. When the right-hand side evaluates to "0" or "false", no scanning is done and the condition fails immediately.

Scanning with SpamAssassin uses a lot of resources. If you scan every message, large ones may cause significant performance degradation. As most spam messages are quite small, it is recommended that you do not scan the big ones. For example:

```
deny message = This message was classified as SPAM
     condition = ${if < {$message_size}{10K}}
     spam = nobody
```

The **spam** condition returns true if the threshold specified in the user's SpamAssassin profile has been matched or exceeded. If you want to use the **spam** condition for its side effects (see the variables below), you can make it always return "true" by appending `:true` to the username.

When the **spam** condition is run, it sets up a number of expansion variables. With the exception of *$spam_score_int*, these are usable only within ACLs; their values are not retained with the message and so cannot be used at delivery time.

$spam_score
: The spam score of the message, for example "3.4" or "30.5". This is useful for inclusion in log or reject messages.

$spam_score_int
: The spam score of the message, multiplied by ten, as an integer value. For example "34" or "305". This is useful for numeric comparisons in conditions. This variable is special; its value is saved with the message, and written to Exim's spool file. This means that it can be used during the whole life of the message on your Exim system, in particular, in routers or transports during the later delivery phase.

$spam_bar
: A string consisting of a number of "+" or "-" characters, representing the integer part of the spam score value. A spam score of 4.4 would have a *$spam_bar* value of "++++". This is useful for inclusion in warning headers, since MUAs can match on such strings.

$spam_report
: A multiline text table, containing the full SpamAssassin report for the message. Useful for inclusion in headers or reject messages.

The **spam** condition caches its results. If you call it again with the same user name, it does not scan again, but rather returns the same values as before.

The **spam** condition returns DEFER if there is any error while running the message through SpamAssassin. If you want to treat DEFER as FAIL (to pass on to the next ACL statement block), append `/defer_ok` to the right-hand side of the spam condition, like this:

```
deny message = This message was classified as SPAM
     spam    = joe/defer_ok
```

This causes messages to be accepted even if there is a problem with **spamd**.

Here is a longer, commented example of the use of the **spam** condition:

```
# put headers in all messages (no matter if spam or not)
warn  spam = nobody:true
```

```
      add_header = X-Spam-Score: $spam_score ($spam_bar)
      add_header = X-Spam-Report: $spam_report

# add second subject line with *SPAM* marker when message
# is over threshold
warn  spam = nobody
      add_header = Subject: *SPAM* $h_Subject:

# reject spam at high scores (> 12)
deny  message = This message scored $spam_score spam points.
      spam = nobody:true
      condition = ${if >{$spam_score_int}{120}{1}{0}}
```

41.4 Scanning MIME parts

The **acl_smtp_mime** global option specifies an ACL that is called once for each MIME part of an SMTP message, including multipart types, in the sequence of their position in the message. Similarly, the **acl_not_smtp_mime** option specifies an ACL that is used for the MIME parts of non-SMTP messages. These options may both refer to the same ACL if you want the same processing in both cases.

These ACLs are called (possibly many times) just before the **acl_smtp_data** ACL in the case of an SMTP message, or just before the **acl_not_smtp** ACL in the case of a non-SMTP message. However, a MIME ACL is called only if the message contains a *Content-Type:* header line. When a call to a MIME ACL does not yield "accept", ACL processing is aborted and the appropriate result code is sent to the client. In the case of an SMTP message, the **acl_smtp_data** ACL is not called when this happens.

You cannot use the **malware** or **spam** conditions in a MIME ACL; these can only be used in the DATA or non-SMTP ACLs. However, you can use the **regex** condition to match against the raw MIME part. You can also use the **mime_regex** condition to match against the decoded MIME part (see section 41.5).

At the start of a MIME ACL, a number of variables are set from the header information for the relevant MIME part. These are described below. The contents of the MIME part are not by default decoded into a disk file except for MIME parts whose content-type is "message/rfc822". If you want to decode a MIME part into a disk file, you can use the **decode** condition. The general syntax is:

```
decode = [/<path>/]<filename>
```

The right hand side is expanded before use. After expansion, the value can be:

(1) "0" or "false", in which case no decoding is done.

(2) The string "default". In that case, the file is put in the temporary "default" directory *<spool_directory>/scan/<message_id>/* with a sequential file name consisting of the message id and a sequence number. The full path and name is available in *$mime_decoded_filename* after decoding.

(3) A full path name starting with a slash. If the full name is an existing directory, it is used as a replacement for the default directory. The filename is then sequentially assigned. If the path does not exist, it is used as the full path and file name.

(4) If the string does not start with a slash, it is used as the filename, and the default path is then used.

The **decode** condition normally succeeds. It is only false for syntax errors or unusual circumstances such as memory shortages. You can easily decode a file with its original, proposed filename using

```
decode = $mime_filename
```

However, you should keep in mind that *$mime_filename* might contain anything. If you place files outside of the default path, they are not automatically unlinked.

For RFC822 attachments (these are messages attached to messages, with a content-type of "message/rfc822"), the ACL is called again in the same manner as for the primary message, only that the *$mime_is_rfc822* expansion variable is set (see below). Attached messages are always decoded to disk before being checked, and the files are unlinked once the check is done.

The MIME ACL supports the **regex** and **mime_regex** conditions. These can be used to match regular expressions against raw and decoded MIME parts, respectively. They are described in section 41.5.

The following list describes all expansion variables that are available in the MIME ACL:

$mime_boundary
If the current part is a multipart (see *$mime_is_multipart*) below, it should have a boundary string, which is stored in this variable. If the current part has no boundary parameter in the *Content-Type:* header, this variable contains the empty string.

$mime_charset
This variable contains the character set identifier, if one was found in the *Content-Type:* header. Examples for charset identifiers are:

```
us-ascii
gb2312 (Chinese)
iso-8859-1
```

Please note that this value is not normalized, so you should do matches case-insensitively.

$mime_content_description
This variable contains the normalized content of the *Content-Description:* header. It can contain a human-readable description of the parts content. Some implementations repeat the filename for attachments here, but they are usually only used for display purposes.

$mime_content_disposition
This variable contains the normalized content of the *Content-Disposition:* header. You can expect strings like "attachment" or "inline" here.

$mime_content_id
This variable contains the normalized content of the *Content-ID:* header. This is a unique ID that can be used to reference a part from another part.

$mime_content_size
This variable is set only after the **decode** modifier (see above) has been successfully run. It contains the size of the decoded part in kilobytes. The size is always rounded up to full kilobytes, so only a completely empty part has a *$mime_content_size* of zero.

$mime_content_transfer_encoding
This variable contains the normalized content of the *Content-transfer-encoding:* header. This is a symbolic name for an encoding type. Typical values are "base64" and "quoted-printable".

$mime_content_type
If the MIME part has a *Content-Type:* header, this variable contains its value, lowercased, and without any options (like "name" or "charset"). Here are some examples of popular MIME types, as they may appear in this variable:

```
text/plain
text/html
application/octet-stream
image/jpeg
audio/midi
```

If the MIME part has no *Content-Type:* header, this variable contains the empty string.

$mime_decoded_filename
This variable is set only after the **decode** modifier (see above) has been successfully run. It contains the full path and file name of the file containing the decoded data.

$mime_filename
This is perhaps the most important of the MIME variables. It contains a proposed filename for an attachment, if one was found in either the *Content-Type:* or *Content-Disposition:* headers. The filename will be RFC2047 decoded, but no additional sanity checks are done. If no filename was found, this variable contains the empty string.

$mime_is_coverletter
This variable attempts to differentiate the "cover letter" of an e-mail from attached data. It can be used to clamp down on flashy or unnecessarily encoded content in the cover letter, while not restricting attachments at all.

The variable contains 1 (true) for a MIME part believed to be part of the cover letter, and 0 (false) for an attachment. At present, the algorithm is as follows:

(1) The outermost MIME part of a message is always a cover letter.

(2) If a multipart/alternative or multipart/related MIME part is a cover letter, so are all MIME subparts within that multipart.

(3) If any other multipart is a cover letter, the first subpart is a cover letter, and the rest are attachments.

(4) All parts contained within an attachment multipart are attachments.

As an example, the following will ban "HTML mail" (including that sent with alternative plain text), while allowing HTML files to be attached. HTML coverletter mail attached to non-HMTL coverletter mail will also be allowed:

```
deny message = HTML mail is not accepted here
!condition = $mime_is_rfc822
condition = $mime_is_coverletter
condition = ${if eq{$mime_content_type}{text/html}{1}{0}}
```

$mime_is_multipart
This variable has the value 1 (true) when the current part has the main type "multipart", for example "multipart/alternative" or "multipart/mixed". Since multipart entities only serve as containers for other parts, you may not want to carry out specific actions on them.

$mime_is_rfc822
This variable has the value 1 (true) if the current part is not a part of the checked message itself, but part of an attached message. Attached message decoding is fully recursive.

$mime_part_count
This variable is a counter that is raised for each processed MIME part. It starts at zero for the very first part (which is usually a multipart). The counter is per-message, so it is reset when processing RFC822 attachments (see *$mime_is_rfc822*). The counter stays set after **acl_smtp_mime** is complete, so you can use it in the DATA ACL to determine the number of MIME parts of a message. For non-MIME messages, this variable contains the value -1.

41.5 Scanning with regular expressions

You can specify your own custom regular expression matches on the full body of the message, or on individual MIME parts.

The **regex** condition takes one or more regular expressions as arguments and matches them against the full message (when called in the DATA ACL) or a raw MIME part (when called in the MIME ACL). The **regex** condition matches linewise, with a maximum line length of 32K characters. That means you cannot have multiline matches with the **regex** condition.

The **mime_regex** condition can be called only in the MIME ACL. It matches up to 32K of decoded content (the whole content at once, not linewise). If the part has not been decoded with the **decode** modifier earlier in the ACL, it is decoded automatically when **mime_regex** is executed (using default path and filename values). If the decoded data is larger than 32K, only the first 32K characters are checked.

The regular expressions are passed as a colon-separated list. To include a literal colon, you must double it. Since the whole right-hand side string is expanded before being used, you must also escape dollar signs and backslashes with more backslashes, or use the `\N` facility to disable expansion. Here is a simple example that contains two regular expressions:

```
deny message = contains blacklisted regex ($regex_match_string)
     regex = [Mm]ortgage : URGENT BUSINESS PROPOSAL
```

The conditions returns true if any one of the regular expressions matches. The *$regex_match_string* expansion variable is then set up and contains the matching regular expression.

Warning: With large messages, these conditions can be fairly CPU-intensive.

41.6 The demime condition

The **demime** ACL condition provides MIME unpacking, sanity checking and file extension blocking. It is usable only in the DATA and non-SMTP ACLs. The **demime** condition uses a simpler interface to MIME decoding than the MIME ACL functionality, but provides no additional facilities. Please note that this condition is deprecated and kept only for backward compatibility. You must set the WITH_OLD_DEMIME option in *Local/Makefile* at build time to be able to use the **demime** condition.

The **demime** condition unpacks MIME containers in the message. It detects errors in MIME containers and can match file extensions found in the message against a list. Using this facility produces files containing the unpacked MIME parts of the message in the temporary scan directory. If you do antivirus scanning, it is recommended that you use the **demime** condition before the antivirus (**malware**) condition.

On the right-hand side of the **demime** condition you can pass a colon-separated list of file extensions that it should match against. For example:

```
deny message = Found blacklisted file attachment
     demime  = vbs:com:bat:pif:prf:lnk
```

If one of the file extensions is found, the condition is true, otherwise it is false. If there is a temporary error while demimeing (for example, "disk full"), the condition defers, and the message is temporarily rejected (unless the condition is on a **warn** verb).

The right-hand side is expanded before being treated as a list, so you can have conditions and lookups there. If it expands to an empty string, "false", or zero ("0"), no demimeing is done and the condition is false.

The **demime** condition set the following variables:

$demime_errorlevel
: When an error is detected in a MIME container, this variable contains the severity of the error, as an integer number. The higher the value, the more severe the error (the current maximum value is 3). If this variable is unset or zero, no error occurred.

$demime_reason
: When *$demime_errorlevel* is greater than zero, this variable contains a human-readable text string describing the MIME error that occurred.

$found_extension
: When the **demime** condition is true, this variable contains the file extension it found.

Both *$demime_errorlevel* and *$demime_reason* are set by the first call of the **demime** condition, and are not changed on subsequent calls.

If you do not want to check for file extensions, but rather use the **demime** condition for unpacking or error checking purposes, pass "*" as the right-hand side value. Here is a more elaborate example of how to use this facility:

```
# Reject messages with serious MIME container errors
deny  message = Found MIME error ($demime_reason).
```

```
      demime = *
      condition = ${if >{$demime_errorlevel}{2}{1}{0}}

# Reject known virus spreading file extensions.
# Accepting these is pretty much braindead.
deny  message = contains $found_extension file (blacklisted).
      demime  = com:vbs:bat:pif:scr

# Freeze .exe and .doc files. Postmaster can
# examine them and eventually thaw them.
deny  log_message = Another $found_extension file.
      demime = exe:doc
      control = freeze
```

42. Adding a local scan function to Exim

In these days of email worms, viruses, and ever-increasing spam, some sites want to apply a lot of checking to messages before accepting them.

The content scanning extension (chapter 41) has facilities for passing messages to external virus and spam scanning software. You can also do a certain amount in Exim itself through string expansions and the **condition** condition in the ACL that runs after the SMTP DATA command or the ACL for non-SMTP messages (see chapter 40), but this has its limitations.

To allow for further customization to a site's own requirements, there is the possibility of linking Exim with a private message scanning function, written in C. If you want to run code that is written in something other than C, you can of course use a little C stub to call it.

The local scan function is run once for every incoming message, at the point when Exim is just about to accept the message. It can therefore be used to control non-SMTP messages from local processes as well as messages arriving via SMTP.

Exim applies a timeout to calls of the local scan function, and there is an option called **local_scan_timeout** for setting it. The default is 5 minutes. Zero means "no timeout". Exim also sets up signal handlers for SIGSEGV, SIGILL, SIGFPE, and SIGBUS before calling the local scan function, so that the most common types of crash are caught. If the timeout is exceeded or one of those signals is caught, the incoming message is rejected with a temporary error if it is an SMTP message. For a non-SMTP message, the message is dropped and Exim ends with a non-zero code. The incident is logged on the main and reject logs.

42.1 Building Exim to use a local scan function

To make use of the local scan function feature, you must tell Exim where your function is before building Exim, by setting LOCAL_SCAN_SOURCE in your *Local/Makefile*. A recommended place to put it is in the *Local* directory, so you might set

```
LOCAL_SCAN_SOURCE=Local/local_scan.c
```

for example. The function must be called *local_scan()*. It is called by Exim after it has received a message, when the success return code is about to be sent. This is after all the ACLs have been run. The return code from your function controls whether the message is actually accepted or not. There is a commented template function (that just accepts the message) in the file _src/local_scan.c_.

If you want to make use of Exim's run time configuration file to set options for your *local_scan()* function, you must also set

```
LOCAL_SCAN_HAS_OPTIONS=yes
```

in *Local/Makefile* (see section 42.3 below).

42.2 API for local_scan()

You must include this line near the start of your code:

```
#include "local_scan.h"
```

This header file defines a number of variables and other values, and the prototype for the function itself. Exim is coded to use unsigned char values almost exclusively, and one of the things this header defines is a shorthand for `unsigned char` called `uschar`. It also contains the following macro definitions, to simplify casting character strings and pointers to character strings:

```
#define CS   (char *)
#define CCS  (const char *)
#define CSS  (char **)
#define US   (unsigned char *)
#define CUS  (const unsigned char *)
#define USS  (unsigned char **)
```

The function prototype for *local_scan()* is:

```
extern int local_scan(int fd, uschar **return_text);
```

The arguments are as follows:

- **fd** is a file descriptor for the file that contains the body of the message (the -D file). The file is open for reading and writing, but updating it is not recommended. **Warning**: You must *not* close this file descriptor.

  The descriptor is positioned at character 19 of the file, which is the first character of the body itself, because the first 19 characters are the message id followed by `-D` and a newline. If you rewind the file, you should use the macro SPOOL_DATA_START_OFFSET to reset to the start of the data, just in case this changes in some future version.

- **return_text** is an address which you can use to return a pointer to a text string at the end of the function. The value it points to on entry is NULL.

The function must return an **int** value which is one of the following macros:

`LOCAL_SCAN_ACCEPT`
: The message is accepted. If you pass back a string of text, it is saved with the message, and made available in the variable *$local_scan_data*. No newlines are permitted (if there are any, they are turned into spaces) and the maximum length of text is 1000 characters.

`LOCAL_SCAN_ACCEPT_FREEZE`
: This behaves as LOCAL_SCAN_ACCEPT, except that the accepted message is queued without immediate delivery, and is frozen.

`LOCAL_SCAN_ACCEPT_QUEUE`
: This behaves as LOCAL_SCAN_ACCEPT, except that the accepted message is queued without immediate delivery.

`LOCAL_SCAN_REJECT`
: The message is rejected; the returned text is used as an error message which is passed back to the sender and which is also logged. Newlines are permitted – they cause a multiline response for SMTP rejections, but are converted to `\n` in log lines. If no message is given, "Administrative prohibition" is used.

`LOCAL_SCAN_TEMPREJECT`
: The message is temporarily rejected; the returned text is used as an error message as for LOCAL_SCAN_REJECT. If no message is given, "Temporary local problem" is used.

`LOCAL_SCAN_REJECT_NOLOGHDR`
: This behaves as LOCAL_SCAN_REJECT, except that the header of the rejected message is not written to the reject log. It has the effect of unsetting the **rejected_header** log selector for just this rejection. If **rejected_header** is already unset (see the discussion of the **log_selection** option in section 49.15), this code is the same as LOCAL_SCAN_REJECT.

`LOCAL_SCAN_TEMPREJECT_NOLOGHDR`
: This code is a variation of LOCAL_SCAN_TEMPREJECT in the same way that LOCAL_SCAN_REJECT_NOLOGHDR is a variation of LOCAL_SCAN_REJECT.

If the message is not being received by interactive SMTP, rejections are reported by writing to **stderr** or by sending an email, as configured by the **-oe** command line options.

42.3 Configuration options for local_scan()

It is possible to have option settings in the main configuration file that set values in static variables in the *local_scan()* module. If you want to do this, you must have the line

```
LOCAL_SCAN_HAS_OPTIONS=yes
```

in your *Local/Makefile* when you build Exim. (This line is in *OS/Makefile-Default*, commented out). Then, in the *local_scan()* source file, you must define static variables to hold the option values, and a table to define them.

The table must be a vector called **local_scan_options**, of type `optionlist`. Each entry is a triplet, consisting of a name, an option type, and a pointer to the variable that holds the value. The entries must appear in alphabetical order. Following **local_scan_options** you must also define a variable called **local_scan_options_count** that contains the number of entries in the table. Here is a short example, showing two kinds of option:

```
static int my_integer_option = 42;
static uschar *my_string_option = US"a default string";

optionlist local_scan_options[] = {
  { "my_integer", opt_int,       &my_integer_option },
  { "my_string",  opt_stringptr, &my_string_option }
};

int local_scan_options_count =
  sizeof(local_scan_options)/sizeof(optionlist);
```

The values of the variables can now be changed from Exim's runtime configuration file by including a local scan section as in this example:

```
begin local_scan
my_integer = 99
my_string = some string of text...
```

The available types of option data are as follows:

opt_bool
: This specifies a boolean (true/false) option. The address should point to a variable of type `BOOL`, which will be set to TRUE or FALSE, which are macros that are defined as "1" and "0", respectively. If you want to detect whether such a variable has been set at all, you can initialize it to TRUE_UNSET. (BOOL variables are integers underneath, so can hold more than two values.)

opt_fixed
: This specifies a fixed point number, such as is used for load averages. The address should point to a variable of type `int`. The value is stored multiplied by 1000, so, for example, 1.4142 is truncated and stored as 1414.

opt_int
: This specifies an integer; the address should point to a variable of type `int`. The value may be specified in any of the integer formats accepted by Exim.

opt_mkint
: This is the same as **opt_int**, except that when such a value is output in a **-bP** listing, if it is an exact number of kilobytes or megabytes, it is printed with the suffix K or M.

opt_octint
: This also specifies an integer, but the value is always interpreted as an octal integer, whether or not it starts with the digit zero, and it is always output in octal.

opt_stringptr
: This specifies a string value; the address must be a pointer to a variable that points to a string (for example, of type `uschar *`).

opt_time
: This specifies a time interval value. The address must point to a variable of type `int`. The value that is placed there is a number of seconds.

If the **-bP** command line option is followed by `local_scan`, Exim prints out the values of all the *local_scan()* options.

42.4 Available Exim variables

The header *local_scan.h* gives you access to a number of C variables. These are the only ones that are guaranteed to be maintained from release to release. Note, however, that you can obtain the value of

any Exim expansion variable, including *$recipients*, by calling *expand_string()*. The exported C variables are as follows:

int body_linecount
This variable contains the number of lines in the message's body.

int body_zerocount
This variable contains the number of binary zero bytes in the message's body.

unsigned int debug_selector
This variable is set to zero when no debugging is taking place. Otherwise, it is a bitmap of debugging selectors. Two bits are identified for use in *local_scan()*; they are defined as macros:

- The `D_v` bit is set when **-v** was present on the command line. This is a testing option that is not privileged – any caller may set it. All the other selector bits can be set only by admin users.
- The `D_local_scan` bit is provided for use by *local_scan()*; it is set by the `+local_scan` debug selector. It is not included in the default set of debugging bits.

Thus, to write to the debugging output only when `+local_scan` has been selected, you should use code like this:

```
if ((debug_selector & D_local_scan) != 0)
  debug_printf("xxx", ...);
```

uschar *expand_string_message
After a failing call to *expand_string()* (returned value NULL), the variable **expand_string_message** contains the error message, zero-terminated.

header_line *header_list
A pointer to a chain of header lines. The **header_line** structure is discussed below.

header_line *header_last
A pointer to the last of the header lines.

uschar *headers_charset
The value of the **headers_charset** configuration option.

BOOL host_checking
This variable is TRUE during a host checking session that is initiated by the **-bh** command line option.

uschar *interface_address
The IP address of the interface that received the message, as a string. This is NULL for locally submitted messages.

int interface_port
The port on which this message was received. When testing with the **-bh** command line option, the value of this variable is -1 unless a port has been specified via the **-oMi** option.

uschar *message_id
This variable contains Exim's message id for the incoming message (the value of *$message_exim_id*) as a zero-terminated string.

uschar *received_protocol
The name of the protocol by which the message was received.

int recipients_count
The number of accepted recipients.

recipient_item *recipients_list
The list of accepted recipients, held in a vector of length **recipients_count**. The **recipient_item** structure is discussed below. You can add additional recipients by calling *receive_add_recipient()* (see below). You can delete recipients by removing them from the vector and adjusting the value in **recipients_count**. In particular, by setting **recipients_count** to zero you remove all recipients. If you then return the value `LOCAL_SCAN_ACCEPT`, the message is accepted, but immediately

blackholed. To replace the recipients, you can set **recipients_count** to zero and then call *receive_add_recipient()* as often as needed.

uschar *sender_address
The envelope sender address. For bounce messages this is the empty string.

uschar *sender_host_address
The IP address of the sending host, as a string. This is NULL for locally-submitted messages.

uschar *sender_host_authenticated
The name of the authentication mechanism that was used, or NULL if the message was not received over an authenticated SMTP connection.

uschar *sender_host_name
The name of the sending host, if known.

int sender_host_port
The port on the sending host.

BOOL smtp_input
This variable is TRUE for all SMTP input, including BSMTP.

BOOL smtp_batched_input
This variable is TRUE for BSMTP input.

int store_pool
The contents of this variable control which pool of memory is used for new requests. See section 42.8 for details.

42.5 Structure of header lines

The **header_line** structure contains the members listed below. You can add additional header lines by calling the *header_add()* function (see below). You can cause header lines to be ignored (deleted) by setting their type to *.

struct header_line *next
A pointer to the next header line, or NULL for the last line.

int type
A code identifying certain headers that Exim recognizes. The codes are printing characters, and are documented in chapter 53 of this manual. Notice in particular that any header line whose type is * is not transmitted with the message. This flagging is used for header lines that have been rewritten, or are to be removed (for example, *Envelope-sender:* header lines.) Effectively, * means "deleted".

int slen
The number of characters in the header line, including the terminating and any internal newlines.

uschar *text
A pointer to the text of the header. It always ends with a newline, followed by a zero byte. Internal newlines are preserved.

42.6 Structure of recipient items

The **recipient_item** structure contains these members:

uschar *address
This is a pointer to the recipient address as it was received.

int pno
This is used in later Exim processing when top level addresses are created by the **one_time** option. It is not relevant at the time *local_scan()* is run and must always contain -1 at this stage.

uschar *errors_to

If this value is not NULL, bounce messages caused by failing to deliver to the recipient are sent to the address it contains. In other words, it overrides the envelope sender for this one recipient. (Compare the **errors_to** generic router option.) If a *local_scan()* function sets an **errors_to** field to an unqualified address, Exim qualifies it using the domain from **qualify_recipient**. When *local_scan()* is called, the **errors_to** field is NULL for all recipients.

42.7 Available Exim functions

The header *local_scan.h* gives you access to a number of Exim functions. These are the only ones that are guaranteed to be maintained from release to release:

pid_t child_open(uschar **argv, uschar **envp, int newumask, int *infdptr, int *outfdptr, BOOL make_leader)

This function creates a child process that runs the command specified by **argv**. The environment for the process is specified by **envp**, which can be NULL if no environment variables are to be passed. A new umask is supplied for the process in **newumask**.

Pipes to the standard input and output of the new process are set up and returned to the caller via the **infdptr** and **outfdptr** arguments. The standard error is cloned to the standard output. If there are any file descriptors "in the way" in the new process, they are closed. If the final argument is TRUE, the new process is made into a process group leader.

The function returns the pid of the new process, or -1 if things go wrong.

int child_close(pid_t pid, int timeout)

This function waits for a child process to terminate, or for a timeout (in seconds) to expire. A timeout value of zero means wait as long as it takes. The return value is as follows:

- >= 0

  The process terminated by a normal exit and the value is the process ending status.

- < 0 and > –256

  The process was terminated by a signal and the value is the negation of the signal number.

- –256

  The process timed out.

- –257

  The was some other error in wait(); **errno** is still set.

pid_t child_open_exim(int *fd)

This function provide you with a means of submitting a new message to Exim. (Of course, you can also call */usr/sbin/sendmail* yourself if you want, but this packages it all up for you.) The function creates a pipe, forks a subprocess that is running

```
exim -t -oem -oi -f <>
```

and returns to you (via the `int *` argument) a file descriptor for the pipe that is connected to the standard input. The yield of the function is the PID of the subprocess. You can then write a message to the file descriptor, with recipients in *To:*, *Cc:*, and/or *Bcc:* header lines.

When you have finished, call *child_close()* to wait for the process to finish and to collect its ending status. A timeout value of zero is usually fine in this circumstance. Unless you have made a mistake with the recipient addresses, you should get a return code of zero.

pid_t child_open_exim2(int *fd, uschar *sender, uschar *sender_authentication)

This function is a more sophisticated version of *child_open()*. The command that it runs is:

```
exim -t -oem -oi -f sender -oMas sender_authentication
```

The third argument may be NULL, in which case the **-oMas** option is omitted.

void debug_printf(char *, ...)
This is Exim's debugging function, with arguments as for *(printf()*. The output is written to the standard error stream. If no debugging is selected, calls to *debug_printf()* have no effect. Normally, you should make calls conditional on the `local_scan` debug selector by coding like this:

```
if ((debug_selector & D_local_scan) != 0)
  debug_printf("xxx", ...);
```

uschar *expand_string(uschar *string)
This is an interface to Exim's string expansion code. The return value is the expanded string, or NULL if there was an expansion failure. The C variable **expand_string_message** contains an error message after an expansion failure. If expansion does not change the string, the return value is the pointer to the input string. Otherwise, the return value points to a new block of memory that was obtained by a call to *store_get()*. See section 42.8 below for a discussion of memory handling.

void header_add(int type, char *format, ...)
This function allows you to an add additional header line at the end of the existing ones. The first argument is the type, and should normally be a space character. The second argument is a format string and any number of substitution arguments as for *sprintf()*. You may include internal newlines if you want, and you must ensure that the string ends with a newline.

void header_add_at_position(BOOL after, uschar *name, BOOL topnot, int type, char *format, ...)
This function adds a new header line at a specified point in the header chain. The header itself is specified as for *header_add()*.

If **name** is NULL, the new header is added at the end of the chain if **after** is true, or at the start if **after** is false. If **name** is not NULL, the header lines are searched for the first non-deleted header that matches the name. If one is found, the new header is added before it if **after** is false. If **after** is true, the new header is added after the found header and any adjacent subsequent ones with the same name (even if marked "deleted"). If no matching non-deleted header is found, the **topnot** option controls where the header is added. If it is true, addition is at the top; otherwise at the bottom. Thus, to add a header after all the *Received:* headers, or at the top if there are no *Received:* headers, you could use

```
header_add_at_position(TRUE, US"Received", TRUE,
  ' ', "X-xxx: ...");
```

Normally, there is always at least one non-deleted *Received:* header, but there may not be if **received_header_text** expands to an empty string.

void header_remove(int occurrence, uschar *name)
This function removes header lines. If **occurrence** is zero or negative, all occurrences of the header are removed. If occurrence is greater than zero, that particular instance of the header is removed. If no header(s) can be found that match the specification, the function does nothing.

BOOL header_testname(header_line *hdr, uschar *name, int length, BOOL notdel)
This function tests whether the given header has the given name. It is not just a string comparison, because white space is permitted between the name and the colon. If the **notdel** argument is true, a false return is forced for all "deleted" headers; otherwise they are not treated specially. For example:

```
if (header_testname(h, US"X-Spam", 6, TRUE)) ...
```

uschar *lss_b64encode(uschar *cleartext, int length)
This function base64-encodes a string, which is passed by address and length. The text may contain bytes of any value, including zero. The result is passed back in dynamic memory that is obtained by calling *store_get()*. It is zero-terminated.

int lss_b64decode(uschar *codetext, uschar **cleartext)
This function decodes a base64-encoded string. Its arguments are a zero-terminated base64-encoded string and the address of a variable that is set to point to the result, which is in dynamic memory. The length of the decoded string is the yield of the function. If the input is invalid base64 data, the yield is -1. A zero byte is added to the end of the output string to make it easy to interpret

as a C string (assuming it contains no zeros of its own). The added zero byte is not included in the returned count.

int lss_match_domain(uschar *domain, uschar *list)
This function checks for a match in a domain list. Domains are always matched caselessly. The return value is one of the following:

`OK`	match succeeded
`FAIL`	match failed
`DEFER`	match deferred

DEFER is usually caused by some kind of lookup defer, such as the inability to contact a database.

int lss_match_local_part(uschar *localpart, uschar *list, BOOL caseless)
This function checks for a match in a local part list. The third argument controls case-sensitivity. The return values are as for *lss_match_domain()*.

int lss_match_address(uschar *address, uschar *list, BOOL caseless)
This function checks for a match in an address list. The third argument controls the case-sensitivity of the local part match. The domain is always matched caselessly. The return values are as for *lss_match_domain()*.

int lss_match_host(uschar *host_name, uschar *host_address, uschar *list)
This function checks for a match in a host list. The most common usage is expected to be

```
lss_match_host(sender_host_name, sender_host_address, ...)
```

An empty address field matches an empty item in the host list. If the host name is NULL, the name corresponding to *$sender_host_address* is automatically looked up if a host name is required to match an item in the list. The return values are as for *lss_match_domain()*, but in addition, *lss_match_host()* returns ERROR in the case when it had to look up a host name, but the lookup failed.

void log_write(unsigned int selector, int which, char *format, ...)
This function writes to Exim's log files. The first argument should be zero (it is concerned with **log_selector**). The second argument can be `LOG_MAIN` or `LOG_REJECT` or `LOG_PANIC` or the inclusive "or" of any combination of them. It specifies to which log or logs the message is written. The remaining arguments are a format and relevant insertion arguments. The string should not contain any newlines, not even at the end.

void receive_add_recipient(uschar *address, int pno)
This function adds an additional recipient to the message. The first argument is the recipient address. If it is unqualified (has no domain), it is qualified with the **qualify_recipient** domain. The second argument must always be -1.

This function does not allow you to specify a private **errors_to** address (as described with the structure of **recipient_item** above), because it pre-dates the addition of that field to the structure. However, it is easy to add such a value afterwards. For example:

```
receive_add_recipient(US"monitor@mydom.example", -1);
recipients_list[recipients_count-1].errors_to =
  US"postmaster@mydom.example";
```

BOOL receive_remove_recipient(uschar *recipient)
This is a convenience function to remove a named recipient from the list of recipients. It returns true if a recipient was removed, and false if no matching recipient could be found. The argument must be a complete email address.

uschar rfc2047_decode(uschar *string, BOOL lencheck, uschar *target, int zeroval, int *lenptr, uschar **error)
This function decodes strings that are encoded according to RFC 2047. Typically these are the contents of header lines. First, each "encoded word" is decoded from the Q or B encoding into a byte-string. Then, if provided with the name of a charset encoding, and if the *iconv()* function is available, an attempt is made to translate the result to the named character set. If this fails, the binary string is returned with an error message.

The first argument is the string to be decoded. If **lencheck** is TRUE, the maximum MIME word length is enforced. The third argument is the target encoding, or NULL if no translation is wanted.

If a binary zero is encountered in the decoded string, it is replaced by the contents of the **zeroval** argument. For use with Exim headers, the value must not be 0 because header lines are handled as zero-terminated strings.

The function returns the result of processing the string, zero-terminated; if **lenptr** is not NULL, the length of the result is set in the variable to which it points. When **zeroval** is 0, **lenptr** should not be NULL.

If an error is encountered, the function returns NULL and uses the **error** argument to return an error message. The variable pointed to by **error** is set to NULL if there is no error; it may be set non-NULL even when the function returns a non-NULL value if decoding was successful, but there was a problem with translation.

int smtp_fflush(void)
This function is used in conjunction with *smtp_printf()*, as described below.

void smtp_printf(char *, ...)
The arguments of this function are like *printf()*; it writes to the SMTP output stream. You should use this function only when there is an SMTP output stream, that is, when the incoming message is being received via interactive SMTP. This is the case when **smtp_input** is TRUE and **smtp_batched_input** is FALSE. If you want to test for an incoming message from another host (as opposed to a local process that used the **-bs** command line option), you can test the value of **sender_host_address**, which is non-NULL when a remote host is involved.

If an SMTP TLS connection is established, *smtp_printf()* uses the TLS output function, so it can be used for all forms of SMTP connection.

Strings that are written by *smtp_printf()* from within *local_scan()* must start with an appropriate response code: 550 if you are going to return LOCAL_SCAN_REJECT, 451 if you are going to return LOCAL_SCAN_TEMPREJECT, and 250 otherwise. Because you are writing the initial lines of a multi-line response, the code must be followed by a hyphen to indicate that the line is not the final response line. You must also ensure that the lines you write terminate with CRLF. For example:

```
smtp_printf("550-this is some extra info\r\n");
return LOCAL_SCAN_REJECT;
```

Note that you can also create multi-line responses by including newlines in the data returned via the **return_text** argument. The added value of using *smtp_printf()* is that, for instance, you could introduce delays between multiple output lines.

The *smtp_printf()* function does not return any error indication, because it does not automatically flush pending output, and therefore does not test the state of the stream. (In the main code of Exim, flushing and error detection is done when Exim is ready for the next SMTP input command.) If you want to flush the output and check for an error (for example, the dropping of a TCP/IP connection), you can call *smtp_fflush()*, which has no arguments. It flushes the output stream, and returns a non-zero value if there is an error.

void *store_get(int)
This function accesses Exim's internal store (memory) manager. It gets a new chunk of memory whose size is given by the argument. Exim bombs out if it ever runs out of memory. See the next section for a discussion of memory handling.

void *store_get_perm(int)
This function is like *store_get()*, but it always gets memory from the permanent pool. See the next section for a discussion of memory handling.

uschar *string_copy(uschar *string)
See below.

uschar *string_copyn(uschar *string, int length)
See below.

uschar *string_sprintf(char *format, ...)

These three functions create strings using Exim's dynamic memory facilities. The first makes a copy of an entire string. The second copies up to a maximum number of characters, indicated by the second argument. The third uses a format and insertion arguments to create a new string. In each case, the result is a pointer to a new string in the current memory pool. See the next section for more discussion.

42.8 More about Exim's memory handling

No function is provided for freeing memory, because that is never needed. The dynamic memory that Exim uses when receiving a message is automatically recycled if another message is received by the same process (this applies only to incoming SMTP connections – other input methods can supply only one message at a time). After receiving the last message, a reception process terminates.

Because it is recycled, the normal dynamic memory cannot be used for holding data that must be preserved over a number of incoming messages on the same SMTP connection. However, Exim in fact uses two pools of dynamic memory; the second one is not recycled, and can be used for this purpose.

If you want to allocate memory that remains available for subsequent messages in the same SMTP connection, you should set

```
store_pool = POOL_PERM
```

before calling the function that does the allocation. There is no need to restore the value if you do not need to; however, if you do want to revert to the normal pool, you can either restore the previous value of **store_pool** or set it explicitly to POOL_MAIN.

The pool setting applies to all functions that get dynamic memory, including *expand_string()*, *store_get()*, and the *string_xxx()* functions. There is also a convenience function called *store_get_perm()* that gets a block of memory from the permanent pool while preserving the value of **store_pool**.

43. System-wide message filtering

The previous chapters (on ACLs and the local scan function) describe checks that can be applied to messages before they are accepted by a host. There is also a mechanism for checking messages once they have been received, but before they are delivered. This is called the *system filter*.

The system filter operates in a similar manner to users' filter files, but it is run just once per message (however many recipients the message has). It should not normally be used as a substitute for routing, because **deliver** commands in a system router provide new envelope recipient addresses. The system filter must be an Exim filter. It cannot be a Sieve filter.

The system filter is run at the start of a delivery attempt, before any routing is done. If a message fails to be completely delivered at the first attempt, the system filter is run again at the start of every retry. If you want your filter to do something only once per message, you can make use of the **first_delivery** condition in an **if** command in the filter to prevent it happening on retries.

Warning: Because the system filter runs just once, variables that are specific to individual recipient addresses, such as *$local_part* and *$domain*, are not set, and the "personal" condition is not meaningful. If you want to run a centrally-specified filter for each recipient address independently, you can do so by setting up a suitable *redirect* router, as described in section 43.8 below.

43.1 Specifying a system filter

The name of the file that contains the system filter must be specified by setting **system_filter**. If you want the filter to run under a uid and gid other than root, you must also set **system_filter_user** and **system_filter_group** as appropriate. For example:

```
system_filter = /etc/mail/exim.filter
system_filter_user = exim
```

If a system filter generates any deliveries directly to files or pipes (via the **save** or **pipe** commands), transports to handle these deliveries must be specified by setting **system_filter_file_transport** and **system_filter_pipe_transport**, respectively. Similarly, **system_filter_reply_transport** must be set to handle any messages generated by the **reply** command.

43.2 Testing a system filter

You can run simple tests of a system filter in the same way as for a user filter, but you should use **-bF** rather than **-bf**, so that features that are permitted only in system filters are recognized.

If you want to test the combined effect of a system filter and a user filter, you can use both **-bF** and **-bf** on the same command line.

43.3 Contents of a system filter

The language used to specify system filters is the same as for users' filter files. It is described in the separate end-user document *Exim's interface to mail filtering*. However, there are some additional features that are available only in system filters; these are described in subsequent sections. If they are encountered in a user's filter file or when testing with **-bf**, they cause errors.

There are two special conditions which, though available in users' filter files, are designed for use in system filters. The condition **first_delivery** is true only for the first attempt at delivering a message, and **manually_thawed** is true only if the message has been frozen, and subsequently thawed by an admin user. An explicit forced delivery counts as a manual thaw, but thawing as a result of the **auto_thaw** setting does not.

Warning: If a system filter uses the **first_delivery** condition to specify an "unseen" (non-significant) delivery, and that delivery does not succeed, it will not be tried again. If you want Exim to retry an unseen delivery until it succeeds, you should arrange to set it up every time the filter runs.

When a system filter finishes running, the values of the variables *$n0 – $n9* are copied into *$sn0 – $sn9* and are thereby made available to users' filter files. Thus a system filter can, for example, set up "scores" to which users' filter files can refer.

43.4 Additional variable for system filters

The expansion variable *$recipients*, containing a list of all the recipients of the message (separated by commas and white space), is available in system filters. It is not available in users' filters for privacy reasons.

43.5 Defer, freeze, and fail commands for system filters

There are three extra commands (**defer**, **freeze** and **fail**) which are always available in system filters, but are not normally enabled in users' filters. (See the **allow_defer**, **allow_freeze** and **allow_fail** options for the *redirect* router.) These commands can optionally be followed by the word **text** and a string containing an error message, for example:

```
fail text "this message looks like spam to me"
```

The keyword **text** is optional if the next character is a double quote.

The **defer** command defers delivery of the original recipients of the message. The **fail** command causes all the original recipients to be failed, and a bounce message to be created. The **freeze** command suspends all delivery attempts for the original recipients. In all cases, any new deliveries that are specified by the filter are attempted as normal after the filter has run.

The **freeze** command is ignored if the message has been manually unfrozen and not manually frozen since. This means that automatic freezing by a system filter can be used as a way of checking out suspicious messages. If a message is found to be all right, manually unfreezing it allows it to be delivered.

The text given with a fail command is used as part of the bounce message as well as being written to the log. If the message is quite long, this can fill up a lot of log space when such failures are common. To reduce the size of the log message, Exim interprets the text in a special way if it starts with the two characters << and contains >> later. The text between these two strings is written to the log, and the rest of the text is used in the bounce message. For example:

```
fail "<<filter test 1>>Your message is rejected \
     because it contains attachments that we are \
     not prepared to receive."
```

Take great care with the **fail** command when basing the decision to fail on the contents of the message, because the bounce message will of course include 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
```

though of course that might let through unwanted bounce messages. The alternative is clever checking of the body and/or headers to detect bounces generated by the filter.

The interpretation of a system filter file ceases after a **defer**, **freeze**, or **fail** command is obeyed. However, any deliveries that were set up earlier in the filter file are honoured, so you can use a sequence such as

```
mail ...
freeze
```

to send a specified message when the system filter is freezing (or deferring or failing) a message. The normal deliveries for the message do not, of course, take place.

43.6 Adding and removing headers in a system filter

Two filter commands that are available only in system filters are:

```
headers add <string>
headers remove <string>
```

The argument for the **headers add** is a string that is expanded and then added to the end of the message's headers. It is the responsibility of the filter maintainer to make sure it conforms to RFC 2822 syntax. Leading white space is ignored, and if the string is otherwise empty, or if the expansion is forced to fail, the command has no effect.

You can use "\n" within the string, followed by white space, to specify continued header lines. More than one header may be added in one command by including "\n" within the string without any following white space. For example:

```
headers add "X-header-1: ....\n  \
             continuation of X-header-1 ...\n\
             X-header-2: ...."
```

Note that the header line continuation white space after the first newline must be placed before the backslash that continues the input string, because white space after input continuations is ignored.

The argument for **headers remove** is a colon-separated list of header names. This command applies only to those headers that are stored with the message; those that are added at delivery time (such as *Envelope-To:* and *Return-Path:*) cannot be removed by this means. If there is more than one header with the same name, they are all removed.

The **headers** command in a system filter makes an immediate change to the set of header lines that was received with the message (with possible additions from ACL processing). Subsequent commands in the system filter operate on the modified set, which also forms the basis for subsequent message delivery. Unless further modified during routing or transporting, this set of headers is used for all recipients of the message.

During routing and transporting, the variables that refer to the contents of header lines refer only to those lines that are in this set. Thus, header lines that are added by a system filter are visible to users' filter files and to all routers and transports. This contrasts with the manipulation of header lines by routers and transports, which is not immediate, but which instead is saved up until the message is actually being written (see section 44.17).

If the message is not delivered at the first attempt, header lines that were added by the system filter are stored with the message, and so are still present at the next delivery attempt. Header lines that were removed are still present, but marked "deleted" so that they are not transported with the message. For this reason, it is usual to make the **headers** command conditional on **first_delivery** so that the set of header lines is not modified more than once.

Because header modification in a system filter acts immediately, you have to use an indirect approach if you want to modify the contents of a header line. For example:

```
headers add "Old-Subject: $h_subject:"
headers remove "Subject"
headers add "Subject: new subject (was: $h_old-subject:)"
headers remove "Old-Subject"
```

43.7 Setting an errors address in a system filter

In a system filter, if a **deliver** command is followed by

```
errors_to <some address>
```

in order to change the envelope sender (and hence the error reporting) for that delivery, any address may be specified. (In a user filter, only the current user's address can be set.) For example, if some mail is being monitored, you might use

```
unseen deliver monitor@spying.example errors_to root@local.example
```

to take a copy which would not be sent back to the normal error reporting address if its delivery failed.

43.8 Per-address filtering

In contrast to the system filter, which is run just once per message for each delivery attempt, it is also possible to set up a system-wide filtering operation that runs once for each recipient address. In this case, variables such as *$local_part* and *$domain* can be used, and indeed, the choice of filter file could be made dependent on them. This is an example of a router which implements such a filter:

```
central_filter:
  check_local_user
  driver = redirect
  domains = +local_domains
  file = /central/filters/$local_part
  no_verify
  allow_filter
  allow_freeze
```

The filter is run in a separate process under its own uid. Therefore, either **check_local_user** must be set (as above), in which case the filter is run as the local user, or the **user** option must be used to specify which user to use. If both are set, **user** overrides.

Care should be taken to ensure that none of the commands in the filter file specify a significant delivery if the message is to go on to be delivered to its intended recipient. The router will not then claim to have dealt with the address, so it will be passed on to subsequent routers to be delivered in the normal way.

44. Message processing

Exim performs various transformations on the sender and recipient addresses of all messages that it handles, and also on the messages' header lines. Some of these are optional and configurable, while others always take place. All of this processing, except rewriting as a result of routing, and the addition or removal of header lines while delivering, happens when a message is received, before it is placed on Exim's queue.

Some of the automatic processing takes place by default only for "locally-originated" messages. This adjective is used to describe messages that are not received over TCP/IP, but instead are passed to an Exim process on its standard input. This includes the interactive "local SMTP" case that is set up by the **-bs** command line option.

Note: Messages received over TCP/IP on the loopback interface (127.0.0.1 or ::1) are not considered to be locally-originated. Exim does not treat the loopback interface specially in any way.

If you want the loopback interface to be treated specially, you must ensure that there are appropriate entries in your ACLs.

44.1 Submission mode for non-local messages

Processing that happens automatically for locally-originated messages (unless **suppress_local_fixups** is set) can also be requested for messages that are received over TCP/IP. The term "submission mode" is used to describe this state. Submission mode is set by the modifier

```
control = submission
```

in a MAIL, RCPT, or pre-data ACL for an incoming message (see sections 40.19 and 40.20). This makes Exim treat the message as a local submission, and is normally used when the source of the message is known to be an MUA running on a client host (as opposed to an MTA). For example, to set submission mode for messages originating on the IPv4 loopback interface, you could include the following in the MAIL ACL:

```
warn  hosts = 127.0.0.1
      control = submission
```

There are some options that can be used when setting submission mode. A slash is used to separate options. For example:

```
control = submission/sender_retain
```

Specifying **sender_retain** has the effect of setting **local_sender_retain** true and **local_from_check** false for the current incoming message. The first of these allows an existing *Sender:* header in the message to remain, and the second suppresses the check to ensure that *From:* matches the authenticated sender. With this setting, Exim still fixes up messages by adding *Date:* and *Message-ID:* header lines if they are missing, but makes no attempt to check sender authenticity in header lines.

When **sender_retain** is not set, a submission mode setting may specify a domain to be used when generating a *From:* or *Sender:* header line. For example:

```
control = submission/domain=some.domain
```

The domain may be empty. How this value is used is described in sections 44.11 and 44.16. There is also a **name** option that allows you to specify the user's full name for inclusion in a created *Sender:* or *From:* header line. For example:

```
accept authenticated = *
       control = submission/domain=wonderland.example/\
                           name=${lookup {$authenticated_id} \
                                  lsearch {/etc/exim/namelist}}
```

Because the name may contain any characters, including slashes, the **name** option must be given last. The remainder of the string is used as the name. For the example above, if */etc/exim/namelist* contains:

```
bigegg:  Humpty Dumpty
```

then when the sender has authenticated as *bigegg*, the generated *Sender:* line would be:

```
Sender: Humpty Dumpty <bigegg@wonderland.example>
```

By default, submission mode forces the return path to the same address as is used to create the *Sender:* header. However, if **sender_retain** is specified, the return path is also left unchanged.

Note: The changes caused by submission mode take effect after the predata ACL. This means that any sender checks performed before the fix-ups use the untrusted sender address specified by the user, not the trusted sender address specified by submission mode. Although this might be slightly unexpected, it does mean that you can configure ACL checks to spot that a user is trying to spoof another's address.

44.2 Line endings

RFC 2821 specifies that CRLF (two characters: carriage-return, followed by linefeed) is the line ending for messages transmitted over the Internet using SMTP over TCP/IP. However, within individual operating systems, different conventions are used. For example, Unix-like systems use just LF, but others use CRLF or just CR.

Exim was designed for Unix-like systems, and internally, it stores messages using the system's convention of a single LF as a line terminator. When receiving a message, all line endings are translated to this standard format. Originally, it was thought that programs that passed messages directly to an MTA within an operating system would use that system's convention. Experience has shown that this is not the case; for example, there are Unix applications that use CRLF in this circumstance. For this reason, and for compatibility with other MTAs, the way Exim handles line endings for all messages is now as follows:

- LF not preceded by CR is treated as a line ending.
- CR is treated as a line ending; if it is immediately followed by LF, the LF is ignored.
- The sequence "CR, dot, CR" does not terminate an incoming SMTP message, nor a local message in the state where a line containing only a dot is a terminator.
- If a bare CR is encountered within a header line, an extra space is added after the line terminator so as not to end the header line. The reasoning behind this is that bare CRs in header lines are most likely either to be mistakes, or people trying to play silly games.
- If the first header line received in a message ends with CRLF, a subsequent bare LF in a header line is treated in the same way as a bare CR in a header line.

44.3 Unqualified addresses

By default, Exim expects every envelope address it receives from an external host to be fully qualified. Unqualified addresses cause negative responses to SMTP commands. However, because SMTP is used as a means of transporting messages from MUAs running on personal workstations, there is sometimes a requirement to accept unqualified addresses from specific hosts or IP networks.

Exim has two options that separately control which hosts may send unqualified sender or recipient addresses in SMTP commands, namely **sender_unqualified_hosts** and **recipient_unqualified_hosts**. In both cases, if an unqualified address is accepted, it is qualified by adding the value of **qualify_domain** or **qualify_recipient**, as appropriate.

Unqualified addresses in header lines are automatically qualified for messages that are locally originated, unless the **-bnq** option is given on the command line. For messages received over SMTP, unqualified addresses in header lines are qualified only if unqualified addresses are permitted in

SMTP commands. In other words, such qualification is also controlled by **sender_unqualified_hosts** and **recipient_unqualified_hosts**,

44.4 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". Examples of two common formats are:

```
From a.oakley@berlin.mus Fri Jan  5 12:35 GMT 1996
From f.butler@berlin.mus Fri, 7 Jan 97 14:00:00 GMT
```

This line precedes the RFC 2822 header lines. For compatibility with Sendmail, Exim recognizes such lines at the start of messages that are submitted to it via the command line (that is, on the standard input). It does not recognize such lines in incoming SMTP messages, unless the sending host matches **ignore_fromline_hosts** or the **-bs** option was used for a local message and **ignore_fromline_local** is set. The recognition is controlled by a regular expression that is defined by the **uucp_from_pattern** option, whose default value matches the two common cases shown above and puts the address that follows "From" into *$1*.

When the caller of Exim for a non-SMTP message that contains a "From" line is a trusted user, the message's sender address is constructed by expanding the contents of **uucp_sender_address**, whose default value is "$1". This is then parsed as an RFC 2822 address. If there is no domain, the local part is qualified with **qualify_domain** unless it is the empty string. However, if the command line **-f** option is used, it overrides the "From" line.

If the caller of Exim is not trusted, the "From" line is recognized, but the sender address is not changed. This is also the case for incoming SMTP messages that are permitted to contain "From" lines.

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, as it is not valid as a header line. This also happens if a "From" line is present in an incoming SMTP message from a source that is not permitted to send them.

44.5 Resent- header lines

RFC 2822 makes provision for sets of header lines starting with the string `Resent-` to be added to a message when it is resent by the original recipient to somebody else. These headers are *Resent-Date:*, *Resent-From:*, *Resent-Sender:*, *Resent-To:*, *Resent-Cc:*, *Resent-Bcc:* and *Resent-Message-ID:*. The RFC says:

> *Resent fields are strictly informational. They MUST NOT be used in the normal processing of replies or other such automatic actions on messages.*

This leaves things a bit vague as far as other processing actions such as address rewriting are concerned. Exim treats **Resent-** header lines as follows:

- A *Resent-From:* line that just contains the login id of the submitting user is automatically rewritten in the same way as *From:* (see below).
- If there's a rewriting rule for a particular header line, it is also applied to **Resent-** header lines of the same type. For example, a rule that rewrites *From:* also rewrites *Resent-From:*.
- For local messages, if *Sender:* is removed on input, *Resent-Sender:* is also removed.
- For a locally-submitted message, if there are any **Resent-** header lines but no *Resent-Date:*, *Resent-From:*, or *Resent-Message-Id:*, they are added as necessary. It is the contents of *Resent-Message-Id:* (rather than *Message-Id:*) which are included in log lines in this case.
- The logic for adding *Sender:* is duplicated for *Resent-Sender:* when any **Resent-** header lines are present.

44.6 The Auto-Submitted: header line

Whenever Exim generates an autoreply, a bounce, or a delay warning message, it includes the header line:

```
Auto-Submitted: auto-replied
```

44.7 The Bcc: header line

If Exim is called with the **-t** option, to take recipient addresses from a message's header, it removes any *Bcc:* header line that may exist (after extracting its addresses). If **-t** is not present on the command line, any existing *Bcc:* is not removed.

44.8 The Date: header line

If a locally-generated or submission-mode message has no *Date:* header line, Exim adds one, using the current date and time, unless the **suppress_local_fixups** control has been specified.

44.9 The Delivery-date: header line

Delivery-date: header lines are not part of the standard RFC 2822 header set. Exim can be configured to add them to the final delivery of messages. (See the generic **delivery_date_add** transport option.) They should not be present in messages in transit. If the **delivery_date_remove** configuration option is set (the default), Exim removes *Delivery-date:* header lines from incoming messages.

44.10 The Envelope-to: header line

Envelope-to: header lines are not part of the standard RFC 2822 header set. Exim can be configured to add them to the final delivery of messages. (See the generic **envelope_to_add** transport option.) They should not be present in messages in transit. If the **envelope_to_remove** configuration option is set (the default), Exim removes *Envelope-to:* header lines from incoming messages.

44.11 The From: header line

If a submission-mode message does not contain a *From:* header line, Exim adds one if either of the following conditions is true:

- The envelope sender address is not empty (that is, this is not a bounce message). The added header line copies the envelope sender address.
- The SMTP session is authenticated and *$authenticated_id* is not empty.

  (1) If no domain is specified by the submission control, the local part is *$authenticated_id* and the domain is *$qualify_domain*.

  (2) If a non-empty domain is specified by the submission control, the local part is *$authenticated_id*, and the domain is the specified domain.

  (3) If an empty domain is specified by the submission control, *$authenticated_id* is assumed to be the complete address.

A non-empty envelope sender takes precedence.

If a locally-generated incoming message does not contain a *From:* header line, and the **suppress_local_fixups** control is not set, Exim adds one containing the sender's address. The calling user's login name and full name are used to construct the address, as described in section 44.18. They are obtained from the password data by calling *getpwuid()* (but see the **unknown_login** configuration option). The address is qualified with **qualify_domain**.

For compatibility with Sendmail, if an incoming, non-SMTP message has a *From:* header line 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 described in section 44.18.

44.12 The Message-ID: header line

If a locally-generated or submission-mode incoming message does not contain a *Message-ID:* or *Resent-Message-ID:* header line, and the **suppress_local_fixups** control is not set, Exim adds a suitable header line to the message. If there are any *Resent-:* headers in the message, it creates *Resent-Message-ID:*. The id is constructed from Exim's internal message id, preceded by the letter E to ensure it starts with a letter, and followed by @ and the primary host name. Additional information can be included in this header line by setting the **message_id_header_text** and/or **message_id_header_domain** options.

44.13 The Received: header line

A *Received:* header line is added at the start of every message. The contents are defined by the **received_header_text** configuration option, and Exim automatically adds a semicolon and a timestamp to the configured string.

The *Received:* header is generated as soon as the message's header lines have been received. At this stage, the timestamp in the *Received:* header line is the time that the message started to be received. This is the value that is seen by the DATA ACL and by the *local_scan()* function.

Once a message is accepted, the timestamp in the *Received:* header line is changed to the time of acceptance, which is (apart from a small delay while the -H spool file is written) the earliest time at which delivery could start.

44.14 The References: header line

Messages created by the *autoreply* transport include a *References:* header line. This is constructed according to the rules that are described in section 3.64 of RFC 2822 (which states that replies should contain such a header line), and section 3.14 of RFC 3834 (which states that automatic responses are not different in this respect). However, because some mail processing software does not cope well with very long header lines, no more than 12 message IDs are copied from the *References:* header line in the incoming message. If there are more than 12, the first one and then the final 11 are copied, before adding the message ID of the incoming message.

44.15 The Return-path: header line

Return-path: header lines are defined as something an MTA may insert when it does the final delivery of messages. (See the generic **return_path_add** transport option.) Therefore, they should not be present in messages in transit. If the **return_path_remove** configuration option is set (the default), Exim removes *Return-path:* header lines from incoming messages.

44.16 The Sender: header line

For a locally-originated message from an untrusted user, Exim may remove an existing *Sender:* header line, and it may add a new one. You can modify these actions by setting the **local_sender_retain** option true, the **local_from_check** option false, or by using the **suppress_local_fixups** control setting.

When a local message is received from an untrusted user and **local_from_check** is true (the default), and the **suppress_local_fixups** control has not been set, a check is made to see if the address given in the *From:* header line is the correct (local) sender of the message. The address that is expected has the login name as the local part and the value of **qualify_domain** as the domain. Prefixes and suffixes for the local part can be permitted by setting **local_from_prefix** and **local_from_suffix** appropriately. If *From:* does not contain the correct sender, a *Sender:* line is added to the message.

If you set **local_from_check** false, this checking does not occur. However, the removal of an existing *Sender:* line still happens, unless you also set **local_sender_retain** to be true. It is not possible to set both of these options true at the same time.

By default, no processing of *Sender:* header lines is done for messages received over TCP/IP or for messages submitted by trusted users. However, when a message is received over TCP/IP in sub-

mission mode, and **sender_retain** is not specified on the submission control, the following processing takes place:

First, any existing *Sender:* lines are removed. Then, if the SMTP session is authenticated, and *$authenticated_id* is not empty, a sender address is created as follows:

- If no domain is specified by the submission control, the local part is *$authenticated_id* and the domain is *$qualify_domain*.
- If a non-empty domain is specified by the submission control, the local part is *$authenticated_id*, and the domain is the specified domain.
- If an empty domain is specified by the submission control, *$authenticated_id* is assumed to be the complete address.

This address is compared with the address in the *From:* header line. If they are different, a *Sender:* header line containing the created address is added. Prefixes and suffixes for the local part in *From:* can be permitted by setting **local_from_prefix** and **local_from_suffix** appropriately.

Note: Whenever a *Sender:* header line is created, the return path for the message (the envelope sender address) is changed to be the same address, except in the case of submission mode when **sender_retain** is specified.

44.17 Adding and removing header lines in routers and transports

When a message is delivered, the addition and removal of header lines can be specified in a system filter, or on any of the routers and transports that process the message. Section 43.6 contains details about modifying headers in a system filter. Header lines can also be added in an ACL as a message is received (see section 40.22).

In contrast to what happens in a system filter, header modifications that are specified on routers and transports apply only to the particular recipient addresses that are being processed by those routers and transports. These changes do not actually take place until a copy of the message is being transported. Therefore, they do not affect the basic set of header lines, and they do not affect the values of the variables that refer to header lines.

Note: In particular, this means that any expansions in the configuration of the transport cannot refer to the modified header lines, because such expansions all occur before the message is actually transported.

For both routers and transports, the result of expanding a **headers_add** option must be in the form of one or more RFC 2822 header lines, separated by newlines (coded as "\n"). For example:

```
headers_add = X-added-header: added by $primary_hostname\n\
              X-added-second: another added header line
```

Exim does not check the syntax of these added header lines.

The result of expanding **headers_remove** must consist of a colon-separated list of header names. This is confusing, because header names themselves are often terminated by colons. In this case, the colons are the list separators, not part of the names. For example:

```
headers_remove = return-receipt-to:acknowledge-to
```

When **headers_add** or **headers_remove** is specified on a router, its value is expanded at routing time, and then associated with all addresses that are accepted by that router, and also with any new addresses that it generates. If an address passes through several routers as a result of aliasing or forwarding, the changes are cumulative.

However, this does not apply to multiple routers that result from the use of the **unseen** option. Any header modifications that were specified by the "unseen" router or its predecessors apply only to the "unseen" delivery.

Addresses that end up with different **headers_add** or **headers_remove** settings cannot be delivered together in a batch, so a transport is always dealing with a set of addresses that have the same header-processing requirements.

The transport starts by writing the original set of header lines that arrived with the message, possibly modified by the system filter. As it writes out these lines, it consults the list of header names that were attached to the recipient address(es) by **headers_remove** options in routers, and it also consults the transport's own **headers_remove** option. Header lines whose names are on either of these lists are not written out. If there are multiple instances of any listed header, they are all skipped.

After the remaining original header lines have been written, new header lines that were specified by routers' **headers_add** options are written, in the order in which they were attached to the address. These are followed by any header lines specified by the transport's **headers_add** option.

This way of handling header line modifications in routers and transports has the following consequences:

- The original set of header lines, possibly modified by the system filter, remains "visible", in the sense that the *$header_xxx* variables refer to it, at all times.
- Header lines that are added by a router's **headers_add** option are not accessible by means of the *$header_xxx* expansion syntax in subsequent routers or the transport.
- Conversely, header lines that are specified for removal by **headers_remove** in a router remain visible to subsequent routers and the transport.
- Headers added to an address by **headers_add** in a router cannot be removed by a later router or by a transport.
- An added header can refer to the contents of an original header that is to be removed, even it has the same name as the added header. For example:

```
headers_remove = subject
headers_add = Subject: new subject (was: $h_subject:)
```

Warning: The **headers_add** and **headers_remove** options cannot be used for a *redirect* router that has the **one_time** option set.

44.18 Constructed addresses

When Exim constructs a sender address for a locally-generated message, it uses the form

<*user name*> <*login*@*qualify_domain*>

For example:

```
Zaphod Beeblebrox <zaphod@end.univ.example>
```

The user name is obtained from the **-F** command line option if set, or otherwise by looking up the calling user by *getpwuid()* 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 upper cased, as is conventional in a number of operating systems. See the **gecos_name** option for a way to tailor the handling of the "gecos" field. The **unknown_username** option can be used to specify user names in cases when there is no password file entry.

In all cases, the user name is made to conform to RFC 2822 by quoting all or parts of it if necessary. In addition, if it contains any non-printing characters, it is encoded as described in RFC 2047, which defines a way of including non-ASCII characters in header lines. The value of the **headers_charset** option specifies the name of the encoding that is used (the characters are assumed to be in this encoding). The setting of **print_topbitchars** controls whether characters with the top bit set (that is, with codes greater than 127) count as printing characters or not.

44.19 Case of local parts

RFC 2822 states that the case of letters in the local parts of addresses cannot be assumed to be non-significant. Exim preserves the case of local parts of addresses, but by default it uses a lower-cased form when it is routing, because on most Unix systems, usernames are in lower case and case-insensitive routing is required. However, any particular router can be made to use the original case for local parts by setting the **caseful_local_part** generic router option.

If you must have mixed-case user names on your system, the best way to proceed, assuming you want case-independent handling of incoming email, is to set up your first router to convert incoming local parts in your domains to the correct case by means of a file lookup. For example:

```
correct_case:
  driver = redirect
  domains = +local_domains
  data = ${lookup{$local_part}cdb\
              {/etc/usercased.cdb}{$value}fail}\
              @$domain
```

For this router, the local part is forced to lower case by the default action (**caseful_local_part** is not set). The lower-cased local part is used to look up a new local part in the correct case. If you then set **caseful_local_part** on any subsequent routers which process your domains, they will operate on local parts with the correct case in a case-sensitive manner.

44.20 Dots in local parts

RFC 2822 forbids empty components in local parts. That is, an unquoted local part may not begin or end with a dot, nor have two consecutive dots in the middle. However, it seems that many MTAs do not enforce this, so Exim permits empty components for compatibility.

44.21 Rewriting addresses

Rewriting of sender and recipient addresses, and addresses in headers, can happen automatically, or as the result of configuration options, as described in chapter 31. The headers that may be affected by this are *Bcc:*, *Cc:*, *From:*, *Reply-To:*, *Sender:*, and *To:*.

Automatic rewriting includes qualification, as mentioned above. The other case in which it can happen is when an incomplete non-local domain is given. The routing process may cause this to be expanded into the full domain name. For example, a header such as

```
To: hare@teaparty
```

might get rewritten as

```
To: hare@teaparty.wonderland.fict.example
```

Rewriting as a result of routing is the one kind of message processing that does not happen at input time, as it cannot be done until the address has been routed.

Strictly, one should not do *any* deliveries of a message until all its addresses have been routed, in case any of the headers get changed as a result of routing. However, doing this in practice would hold up many deliveries for unreasonable amounts of time, just because one address could not immediately be routed. Exim therefore does not delay other deliveries when routing of one or more addresses is deferred.

45. SMTP processing

Exim supports a number of different ways of using the SMTP protocol, and its LMTP variant, which is an interactive protocol for transferring messages into a closed mail store application. This chapter contains details of how SMTP is processed. For incoming mail, the following are available:

- SMTP over TCP/IP (Exim daemon or *inetd*);
- SMTP over the standard input and output (the **-bs** option);
- Batched SMTP on the standard input (the **-bS** option).

For mail delivery, the following are available:

- SMTP over TCP/IP (the *smtp* transport);
- LMTP over TCP/IP (the *smtp* transport with the **protocol** option set to "lmtp");
- LMTP over a pipe to a process running in the local host (the *lmtp* transport);
- Batched SMTP to a file or pipe (the *appendfile* and *pipe* transports with the **use_bsmtp** option set).

Batched SMTP is the name for a process in which batches of messages are stored in or read from files (or pipes), in a format in which SMTP commands are used to contain the envelope information.

45.1 Outgoing SMTP and LMTP over TCP/IP

Outgoing SMTP and LMTP over TCP/IP is implemented by the *smtp* transport. The **protocol** option selects which protocol is to be used, but the actual processing is the same in both cases.

If, in response to its EHLO command, Exim is told that the SIZE parameter is supported, it 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 the remote server advertises support for PIPELINING, Exim uses the pipelining extension to SMTP (RFC 2197) to reduce the number of TCP/IP packets required for the transaction.

If the remote server advertises support for the STARTTLS command, and Exim was built to support TLS encryption, it tries to start a TLS session unless the server matches **hosts_avoid_tls**. See chapter 39 for more details.

If the remote server advertises support for the AUTH command, Exim scans the authenticators configuration for any suitable client settings, as described in chapter 33.

Responses from the remote host are supposed to be terminated by CR followed by LF. However, there are known to be hosts that do not send CR characters, so in order to be able to interwork with such hosts, Exim treats LF on its own as a line terminator.

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 setting of the **max_rcpts** option in the *smtp* transport allows, in which 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 host name 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 host name plus IP address, so if one address of a multi-homed host is broken, it will soon be skipped most of the time. See the next section for more detail about error handling.

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 so that it can deliver another message using the same socket. 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.

The **connection_max_messages** 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.

45.2 Errors in outgoing SMTP

Three different kinds of error are recognized for outgoing SMTP: host errors, message errors, and recipient errors.

Host errors

A host error is not associated with a particular message or with a particular recipient of a message. The host errors are:

- Connection refused or timed out,
- Any error response code on connection,
- Any error response code to EHLO or HELO,
- Loss of connection at any time, except after ".",
- I/O errors at any time,
- Timeouts during the session, other than in response to MAIL, RCPT or the "." at the end of the data.

For a host error, a permanent error response on connection, or in response to EHLO, causes all addresses routed to the host to be failed. Any other host error causes all addresses to be deferred, and retry data to be created for the host. It is not tried again, for any message, until its retry time arrives. If the current set of addresses are not all delivered in this run (to some alternative host), the message is added to the list of those waiting for this host, so if it is still undelivered when a subsequent successful delivery is made to the host, it will be sent 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:

- Any error response code to MAIL, DATA, or the "." that terminates the data,
- Timeout after MAIL,
- Timeout or loss of connection after the "." 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 message error, a permanent error response (5*xx*) causes all addresses to be failed, and a delivery error report to be returned to the sender. A temporary error response (4*xx*), or one of the timeouts, causes all addresses to be deferred. Retry data is not created for the host, but instead, a retry record for the combination of host plus 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, it will not stop the delivery of other mail.

If the remote host specified support for the SIZE parameter in its response to EHLO, Exim adds SIZE=*nnn* to the MAIL command, so an over-large message will cause a message error because the error arrives as a response to MAIL.

Recipient errors

A recipient error is associated with a particular recipient of a message. The recipient errors are:

- Any error response to RCPT,
- Timeout after RCPT.

For a recipient error, a permanent error response (5*xx*) causes the recipient address to be failed, and a bounce message to be returned to the sender. A temporary error response (4*xx*) or a timeout causes the failing address to be deferred, and routing retry data to be created for it. This is used to delay processing of the address in subsequent queue runs, until its routing retry time arrives. This applies to all messages, but because it operates only in queue runs, one attempt will be 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 getting delivered. If a delivery to the address does succeed, the retry information gets cleared, so all stuck messages get tried again, and the retry clock is reset.

The message is not added to the list of those waiting for this host. Use of the host for other messages 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 get 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.

In all cases, 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. In other words, recipient error retry data does not take effect until the next delivery attempt.

Some hosts have been observed to give temporary error responses to every MAIL command at certain times ("insufficient space" has been seen). It would be nice if such circumstances could be recognized, and defer 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. Exim makes this assumption, and treats them as if a temporary error response had been received. A timeout after "." 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, it is 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 "." 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 dialogue, causing it then to be treated as a host error.

There is experimental evidence that some MTAs drop the connection after the terminating "." if they do not like the contents of the message for some reason, 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 rather than a host error, in order not to delay other messages to the same host.

45.3 Incoming SMTP messages over TCP/IP

Incoming SMTP messages 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 /opt/exim/bin/exim in.exim -bs
```

Exim distinguishes between this case and the case of a locally running user agent using the **-bs** option by checking whether or not the standard input is a socket. When it is, either the port must be privileged (less than 1024), or the caller must be root or the Exim user. If any other user passes a

socket with an unprivileged port number, Exim prints a message on the standard error stream and exits with an error code.

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 **smtp_connection** log selector.

Commands from the remote host are supposed to be terminated by CR followed by LF. However, there are known to be hosts that do not send CR characters. In order to be able to interwork with such hosts, Exim treats LF on its own as a line terminator. Furthermore, because common code is used for receiving messages from all sources, a CR on its own is also interpreted as a line terminator. However, the sequence "CR, dot, CR" does not terminate incoming SMTP data.

One area that sometimes gives rise to problems concerns the EHLO or HELO commands. Some clients send syntactically invalid versions of these commands, which Exim rejects by default. (This is nothing to do with verifying the data that is sent, so **helo_verify_hosts** is not relevant.) You can tell Exim not to apply a syntax check by setting **helo_accept_junk_hosts** to match the broken hosts that send invalid commands.

The amount of disk space 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. If **check_spool_space** is set, the check is for that amount of 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 "." that terminates the data. If the remote host logs this text it can help with tracing what has happened to a message.

The Exim daemon can limit the number of simultaneous incoming connections it is prepared to handle (see the **smtp_accept_max** option). It can also limit the number of simultaneous incoming connections from a single remote host (see the **smtp_accept_max_per_host** option). Additional connection attempts are rejected using the SMTP temporary error code 421.

The Exim daemon does not rely on the SIGCHLD signal to detect when a subprocess has finished, as this can get lost at busy times. Instead, it looks for completed subprocesses every time it wakes up. Provided there are other things happening (new incoming calls, starts of queue runs), completed processes will be noticed and tidied away. On very quiet systems you may sometimes see a "defunct" Exim process hanging about. This is not a problem; it will be noticed when the daemon next wakes up.

When running as a daemon, Exim can reserve some SMTP slots for specific hosts, and can also be set up to reject SMTP calls from non-reserved hosts at times of high system load – for details see the **smtp_accept_reserve**, **smtp_load_reserve**, and **smtp_reserve_hosts** options. The load check applies in both the daemon and *inetd* cases.

Exim normally starts a delivery process for each message received, though this can be varied by means of the **-odq** command line option and the **queue_only**, **queue_only_file**, and **queue_only_load** options. The number of simultaneously running delivery processes started in this way from SMTP input can be limited by the **smtp_accept_queue** and **smtp_accept_queue_per_connection** options. When either limit is reached, subsequently received messages are just put on the input queue without starting a delivery process.

The controls that involve counts of incoming SMTP calls (**smtp_accept_max**, **smtp_accept_queue**, **smtp_accept_reserve**) are not available when Exim is started up from the *inetd* daemon, because in that case each connection is handled by an entirely independent Exim process. Control by load average is, however, available with *inetd*.

Exim can be configured to verify addresses in incoming SMTP commands as they are received. See chapter 40 for details. It can also be configured to rewrite addresses at this time – before any syntax checking is done. See section 31.9.

Exim can also be configured to limit the rate at which a client host submits MAIL and RCPT commands in a single SMTP session. See the **smtp_ratelimit_hosts** option.

45.4 Unrecognized SMTP commands

If Exim receives more than **smtp_max_unknown_commands** unrecognized SMTP commands during a single SMTP connection, it drops the connection after sending the error response to the last command. The default value for **smtp_max_unknown_commands** is 3. This is a defence against some kinds of abuse that subvert web servers into making connections to SMTP ports; in these circumstances, a number of non-SMTP lines are sent first.

45.5 Syntax and protocol errors in SMTP commands

A syntax error is detected if an SMTP command is recognized, but there is something syntactically wrong with its data, for example, a malformed email address in a RCPT command. Protocol errors include invalid command sequencing such as RCPT before MAIL. If Exim receives more than **smtp_max_synprot_errors** such commands during a single SMTP connection, it drops the connection after sending the error response to the last command. The default value for **smtp_max_synprot_errors** is 3. This is a defence against broken clients that loop sending bad commands (yes, it has been seen).

45.6 Use of non-mail SMTP commands

The "non-mail" SMTP commands are those other than MAIL, RCPT, and DATA. Exim counts such commands, and drops the connection if there are too many of them in a single SMTP session. This action catches some denial-of-service attempts and things like repeated failing AUTHs, or a mad client looping sending EHLO. The global option **smtp_accept_max_nonmail** defines what "too many" means. Its default value is 10.

When a new message is expected, one occurrence of RSET is not counted. This allows a client to send one RSET between messages (this is not necessary, but some clients do it). Exim also allows one uncounted occurrence of HELO or EHLO, and one occurrence of STARTTLS between messages. After starting up a TLS session, another EHLO is expected, and so it too is not counted.

The first occurrence of AUTH in a connection, or immediately following STARTTLS is also not counted. Otherwise, all commands other than MAIL, RCPT, DATA, and QUIT are counted.

You can control which hosts are subject to the limit set by **smtp_accept_max_nonmail** by setting **smtp_accept_max_nonmail_hosts**. The default value is `*`, which makes the limit apply to all hosts. This option means that you can exclude any specific badly-behaved hosts that you have to live with.

45.7 The VRFY and EXPN commands

When Exim receives a VRFY or EXPN command on a TCP/IP connection, it runs the ACL specified by **acl_smtp_vrfy** or **acl_smtp_expn** (as appropriate) in order to decide whether the command should be accepted or not. If no ACL is defined, the command is rejected.

When VRFY is accepted, it runs exactly the same code as when Exim is called with the **-bv** option.

When EXPN is accepted, 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.

45.8 The ETRN command

RFC 1985 describes an SMTP command called ETRN that is designed to overcome the security problems of the TURN command (which has fallen into disuse). When Exim receives an ETRN command on a TCP/IP connection, it runs the ACL specified by **acl_smtp_etrn** in order to decide whether the command should be accepted or not. If no ACL is defined, the command is rejected.

The ETRN command is concerned with "releasing" messages that are awaiting delivery to certain hosts. As Exim does not organize its message queue by host, the only form of ETRN that is supported by default is the one where the text starts with the "#" prefix, in which case the remainder of the text

is specific to the SMTP server. A valid ETRN command causes a run of Exim with the **-R** option to happen, 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". When **smtp_etrn_serialize** is set (the default), Exim 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 the process completes. However, Exim does not keep the SMTP session waiting for the ETRN process to complete. Once ETRN is accepted, the client is sent a "success" return code. Obviously there is scope for hints records to get 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.

For more control over what ETRN does, the **smtp_etrn_command** option can 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 which are independently expanded. The expansion variable *$domain* is set to the argument of the ETRN command, and no syntax checking is done on the contents of this argument. Exim does not wait for the command to complete, so its status code is not checked. Exim runs under its own uid and gid when receiving incoming SMTP, so it is not possible for it to change them before running the command.

45.9 Incoming 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** option. This form of SMTP is handled in the same way as incoming messages over TCP/IP (including the use of ACLs), except that the envelope sender given in a MAIL command is ignored unless the caller is trusted. In an ACL you can detect this form of SMTP input by testing for an empty host identification. It is common to have this as the first line in the ACL that runs for RCPT commands:

```
accept hosts = :
```

This accepts SMTP messages from local processes without doing any other tests.

45.10 Outgoing batched SMTP

Both the *appendfile* and *pipe* transports can be used for handling batched SMTP. Each has an option called **use_bsmtp** which causes messages to be output in BSMTP format. No SMTP responses are possible for this form of delivery. All it is doing is using SMTP commands as a way of transmitting the envelope along with the message.

The message is written to the file or pipe preceded by the SMTP commands MAIL and RCPT, and followed by a line containing a single dot. Lines in the message that start with a dot have an extra dot added. The SMTP command HELO is not normally used. If it is required, the **message_prefix** option can be used to specify it.

Because *appendfile* and *pipe* are both local transports, they accept only one recipient address at a time by default. However, you can arrange for them to handle several addresses at once by setting the **batch_max** option. When this is done for BSMTP, messages may contain multiple RCPT commands. See chapter 25 for more details.

When one or more addresses are routed to a BSMTP transport by a router that sets up a host list, the name of the first host on the list is available to the transport in the variable *$host*. Here is an example of such a transport and router:

```
begin routers
route_append:
  driver = manualroute
  transport = smtp_appendfile
  route_list = domain.example  batch.host.example

begin transports
smtp_appendfile:
  driver = appendfile
  directory = /var/bsmtp/$host
  batch_max = 1000
  use_bsmtp
  user = exim
```

This causes messages addressed to *domain.example* to be written in BSMTP format to */var/bsmtp/batch.host.example*, with only a single copy of each message (unless there are more than 1000 recipients).

45.11 Incoming batched SMTP

The **-bS** command line option causes Exim to accept one or more messages by reading SMTP on the standard input, but to 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 just get qualified. HELO and EHLO act as RSET; VRFY, EXPN, ETRN and HELP, act as NOOP; QUIT quits.

No policy checking is done for BSMTP input. That is, no ACL is run at anytime. In this respect it is like non-SMTP local input.

If an error is detected while reading a message, including a missing "." 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.

46. Customizing bounce and warning messages

When a message fails to be delivered, or remains on the queue for more than a configured amount of time, Exim sends a message to the original sender, or to an alternative configured address. The text of these messages is built into the code of Exim, but it is possible to change it, either by adding a single string, or by replacing each of the paragraphs by text supplied in a file.

The *From:* and *To:* header lines are automatically generated; you can cause a *Reply-To:* line to be added by setting the **errors_reply_to** option. Exim also adds the line

```
Auto-Submitted: auto-generated
```

to all warning and bounce messages,

46.1 Customizing bounce messages

If **bounce_message_text** is set, its contents are included in the default message immediately after "This message was created automatically by mail delivery software." The string is not expanded. It is not used if **bounce_message_file** is set.

When **bounce_message_file** is set, it must point to 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 which can be of use here: *$bounce_recipient* is set to the recipient of an error message while it is being created, and *$bounce_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 headers, and should include at least a *Subject:* header. Exim does not check the syntax of these headers.
- 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 (**bounce_message_file** unset) is equivalent to the following file, in which the sixth item is empty. The *Subject:* and some other lines have been split in order to fit them on the page:

```
Subject: Mail delivery failed
  ${if eq{$sender_address}{$bounce_recipient}
  {: returning message to sender}}
****
This message was created automatically by mail delivery software.

A message ${if eq{$sender_address}{$bounce_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
------ $bounce_return_size_limit or so are included here.
****
```

46.2 Customizing warning messages

The option **warn_message_file** can be pointed at a template file for use when warnings about message delays are created. In this case there are only three text sections:

- The first item is included in the headers, and should include at least a *Subject:* header. Exim does not check the syntax of these headers.
- The second item forms the start of the warning message. After it, Exim lists the delayed addresses.
- The third item then ends the message.

The default state is equivalent to the following file, except that some lines have been split here, in order to fit them on the page:

```
Subject: Warning: message $message_exim_id delayed
  $warn_message_delay
****
This message was created automatically by mail delivery software.

A message ${if eq{$sender_address}{$warn_message_recipients}
{that you sent }{sent by

<$sender_address>

}}has not been delivered to all of its recipients after
more than $warn_message_delay on the queue on $primary_hostname.

The message identifier is:     $message_exim_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.
```

However, in the default state the subject and date lines are omitted if no appropriate headers exist. During the expansion of this file, *$warn_message_delay* is set to the delay time in one of the forms "<*n*> minutes" or "<*n*> hours", and *$warn_message_recipients* contains a list of recipients for the warning message. There may be more than one if there are multiple addresses with different **errors_to** settings on the routers that handled them.

47. Some common configuration settings

This chapter discusses some configuration settings that seem to be fairly common. More examples and discussion can be found in the Exim book.

47.1 Sending mail to a smart host

If you want to send all mail for non-local domains to a "smart host", you should replace the default *dnslookup* router with a router which does the routing explicitly:

```
send_to_smart_host:
  driver = manualroute
  route_list = !+local_domains smart.host.name
  transport = remote_smtp
```

You can use the smart host's IP address instead of the name if you wish. If you are using Exim only to submit messages to a smart host, and not for receiving incoming messages, you can arrange for it to do the submission synchronously by setting the **mua_wrapper** option (see chapter 48).

47.2 Using Exim to handle mailing lists

Exim can be used to run simple mailing lists, but for large and/or complicated requirements, the use of additional specialized mailing list software such as Majordomo or Mailman is recommended.

The *redirect* router can be used to handle mailing lists where each list is maintained in a separate file, which can therefore be managed by an independent manager. The **domains** router option can be used to run these lists in a separate domain from normal mail. For example:

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

This router is skipped for domains other than *lists.example*. For addresses in that domain, it looks for a file that matches the local part. If there is no such file, the router declines, but because **no_more** is set, no subsequent routers are tried, and so the whole delivery fails.

The **forbid_pipe** and **forbid_file** options prevent a local part from being expanded into a file name or a pipe delivery, which is usually inappropriate in 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. However, before acting on this, Exim verifies the error address, and ignores it if verification fails.

For example, using the configuration above, mail sent to *dicts@lists.example* is passed on to those addresses contained in */usr/lists/dicts*, with error reports directed to *dicts-request@lists.example*, provided that this address can be verified. There could be a file called */usr/lists/dicts-request* containing the address(es) of this particular list's manager(s), but other approaches, such as setting up an earlier router (possibly using the **local_part_prefix** or **local_part_suffix** options) to handle addresses of the form **owner-***xxx* or **xxx-***request*, are also possible.

47.3 Syntax errors in mailing lists

If an entry in redirection data contains a syntax error, Exim normally defers delivery of the original address. That means that a syntax error in a mailing list holds up all deliveries to the list. This may not be appropriate when a list is being maintained automatically from data supplied by users, and the addresses are not rigorously checked.

If the **skip_syntax_errors** option is set, the *redirect* router just skips entries that fail to parse, noting the incident in the log. If in addition **syntax_errors_to** is set to a verifiable address, a message is sent to it whenever a broken address is skipped. It is usually appropriate to set **syntax_errors_to** to the same address as **errors_to**.

47.4 Re-expansion of mailing lists

Exim remembers every individual address to which a message has been delivered, in order to avoid duplication, 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 re-expanded when the delivery is next tried. This means that alterations to the list are taken into account at each delivery attempt, so addresses that have been added to the list since the message arrived will therefore receive a copy of the message, even though it pre-dates their subscription.

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

The original top-level address is remembered with each of the generated addresses, and is output in any log messages. However, any intermediate parent addresses are not recorded. This makes a difference to the log only if the **all_parents** selector is set, but for mailing lists there is normally only one level of expansion anyway.

47.5 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 generic **senders** option to restrict the router that handles the list.

The following example uses the same file as a list of recipients and as a list of permitted senders. It requires three routers:

```
lists_request:
  driver = redirect
  domains = lists.example
  local_part_suffix = -request
  file = /usr/lists/$local_part$local_part_suffix
  no_more

lists_post:
  driver = redirect
  domains = lists.example
  senders = ${if exists {/usr/lists/$local_part}\
              {lsearch;/usr/lists/$local_part}{*}}
  file = /usr/lists/$local_part
  forbid_pipe
  forbid_file
  errors_to = $local_part-request@lists.example
  no_more

lists_closed:
  driver = redirect
  domains = lists.example
  allow_fail
  data = :fail: $local_part@lists.example is a closed mailing list
```

All three routers have the same **domains** setting, so for any other domains, they are all skipped. The first router runs only if the local part ends in **-request**. It handles messages to the list manager(s) by means of an open mailing list.

The second router runs only if the **senders** precondition is satisfied. It checks for the existence of a list that corresponds to the local part, and then checks that the sender is on the list by means of a linear search. It is necessary to check for the existence of the file before trying to search it, because otherwise Exim thinks there is a configuration error. If the file does not exist, the expansion of **senders** is *, which matches all senders. This means that the router runs, but because there is no list, declines, and **no_more** ensures that no further routers are run. The address fails with an "unrouteable address" error.

The third router runs only if the second router is skipped, which happens when a mailing list exists, but the sender is not on it. This router forcibly fails the address, giving a suitable error message.

47.6 Variable Envelope Return Paths (VERP)

Variable Envelope Return Paths – see **http://cr.yp.to/proto/verp.txt** – are a way of helping mailing list administrators discover which subscription address is the cause of a particular delivery failure. The idea is to encode the original recipient address in the outgoing envelope sender address, so that if the message is forwarded by another host and then subsequently bounces, the original recipient can be extracted from the recipient address of the bounce.

Envelope sender addresses can be modified by Exim using two different facilities: the **errors_to** option on a router (as shown in previous mailing list examples), or the **return_path** option on a transport. The second of these is effective only if the message is successfully delivered to another host; it is not used for errors detected on the local host (see the description of **return_path** in chapter 24). Here is an example of the use of **return_path** to implement VERP on an *smtp* transport:

```
verp_smtp:
  driver = smtp
  max_rcpt = 1
  return_path = \
    ${if match {$return_path}{^(.+?)-request@your.dom.example\$}\
      {$1-request+$local_part=$domain@your.dom.example}fail}
```

This has the effect of rewriting the return path (envelope sender) on outgoing SMTP messages, if the local part of the original return path ends in "-request", and the domain is *your.dom.example*. The rewriting inserts the local part and domain of the recipient into the return path. Suppose, for example, that a message whose return path has been set to *somelist-request@your.dom.example* is sent to *subscriber@other.dom.example*. In the transport, the return path is rewritten as

```
somelist-request+subscriber=other.dom.example@your.dom.example
```

For this to work, you must tell Exim to send multiple copies of messages that have more than one recipient, so that each copy has just one recipient. This is achieved by setting **max_rcpt** to 1. Without this, a single copy of a message might be sent to several different recipients in the same domain, in which case *$local_part* is not available in the transport, because it is not unique.

Unless your host is doing nothing but mailing list deliveries, you should probably use a separate transport for the VERP deliveries, so as not to use extra resources in making one-per-recipient copies for other deliveries. This can easily be done by expanding the **transport** option in the router:

```
dnslookup:
  driver = dnslookup
  domains = ! +local_domains
  transport = \
    ${if match {$return_path}{^(.+?)-request@your.dom.example\$}\
      {verp_smtp}{remote_smtp}}
  no_more
```

If you want to change the return path using **errors_to** in a router instead of using **return_path** in the transport, you need to set **errors_to** on all routers that handle mailing list addresses. This will ensure that all delivery errors, including those detected on the local host, are sent to the VERP address.

On a host that does no local deliveries and has no manual routing, only the *dnslookup* router needs to be changed. A special transport is not needed for SMTP deliveries. Every mailing list recipient has its own return path value, and so Exim must hand them to the transport one at a time. Here is an example of a *dnslookup* router that implements VERP:

```
verp_dnslookup:
  driver = dnslookup
  domains = ! +local_domains
  transport = remote_smtp
  errors_to = \
    ${if match {$return_path}{^(.+?)-request@your.dom.example\$}}
     {$1-request+$local_part=$domain@your.dom.example}fail}
  no_more
```

Before you start sending out messages with VERPed return paths, you must also configure Exim to accept the bounce messages that come back to those paths. Typically this is done by setting a **local_part_suffix** option for a router, and using this to route the messages to wherever you want to handle them.

The overhead incurred in using VERP depends very much 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).

47.7 Virtual domains

The phrase *virtual domain* is unfortunately used with two rather different meanings:

- A domain for which there are no real mailboxes; all valid local parts are aliases for other email addresses. Common examples are organizational top-level domains and "vanity" domains.
- One of a number of independent domains that are all handled by the same host, with mailboxes on that host, but where the mailbox owners do not necessarily have login accounts on that host.

The first usage is probably more common, and does seem more "virtual" than the second. This kind of domain can be handled in Exim with a straightforward aliasing router. One approach is to create a separate alias file for each virtual domain. Exim can test for the existence of the alias file to determine whether the domain exists. The *dsearch* lookup type is useful here, leading to a router of this form:

```
virtual:
  driver = redirect
  domains = dsearch;/etc/mail/virtual
  data = ${lookup{$local_part}lsearch{/etc/mail/virtual/$domain}}
  no_more
```

The **domains** option specifies that the router is to be skipped, unless there is a file in the */etc/mail/virtual* directory whose name is the same as the domain that is being processed. When the router runs, it looks up the local part in the file to find a new address (or list of addresses). The **no_more** setting ensures that if the lookup fails (leading to **data** being an empty string), Exim gives up on the address without trying any subsequent routers.

This one router can handle all the virtual domains because the alias file names follow a fixed pattern. Permissions can be arranged so that appropriate people can edit the different alias files. A successful aliasing operation results in a new envelope recipient address, which is then routed from scratch.

The other kind of "virtual" domain can also be handled in a straightforward way. One approach is to create a file for each domain containing a list of valid local parts, and use it in a router like this:

```
my_domains:
  driver = accept
  domains = dsearch;/etc/mail/domains
  local_parts = lsearch;/etc/mail/domains/$domain
  transport = my_mailboxes
```

The address is accepted if there is a file for the domain, and the local part can be found in the file. The **domains** option is used to check for the file's existence because **domains** is tested before the **local_parts** option (see section 3.12). You cannot use **require_files**, because that option is tested after **local_parts**. The transport is as follows:

```
my_mailboxes:
  driver = appendfile
  file = /var/mail/$domain/$local_part
  user = mail
```

This uses a directory of mailboxes for each domain. The **user** setting is required, to specify which uid is to be used for writing to the mailboxes.

The configuration shown here is just one example of how you might support this requirement. There are many other ways this kind of configuration can be set up, for example, by using a database instead of separate files to hold all the information about the domains.

47.8 Multiple user mailboxes

Heavy email users often want to operate with multiple mailboxes, into which incoming mail is automatically sorted. A popular way of handling this is to allow users to use multiple sender addresses, so that replies can easily be identified. Users are permitted to add prefixes or suffixes to their local parts for this purpose. The wildcard facility of the generic router options **local_part_prefix** and **local_part_suffix** can be used for this. For example, consider this router:

```
userforward:
  driver = redirect
  check_local_user
  file = $home/.forward
  local_part_suffix = -*
  local_part_suffix_optional
  allow_filter
```

It runs a user's *.forward* file for all local parts of the form *username*-*. Within the 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 fall through to subsequent routers, and, assuming no subsequent use of the **local_part_suffix** option is made, they presumably fail. Thus, users have control over which suffixes are valid.

Alternatively, a suffix can be used to trigger the use of a different *.forward* file – which is the way a similar facility is implemented in another MTA:

```
userforward:
  driver = redirect
  check_local_user
  file = $home/.forward$local_part_suffix
  local_part_suffix = -*
  local_part_suffix_optional
  allow_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 passed on to subsequent routers, which could, if required, look for an unqualified *.forward* file to use as a default.

47.9 Simplified vacation processing

The traditional way of running the *vacation* program is for a user to set up a pipe command in a *.forward* file (see section 22.6 for syntax details). This is prone to error by inexperienced users. There are two features of Exim that can be used to make this process simpler for users:

- A local part prefix such as "vacation-" can be specified on a router which can cause the message to be delivered directly to the *vacation* program, or alternatively can use Exim's *autoreply* transport. The contents of a user's *.forward* file are then much simpler. For example:

  ```
  spqr, vacation-spqr
  ```

- The **require_files** generic router option can be used to trigger a vacation delivery by checking for the existence of a certain file in the user's home directory. The **unseen** generic option should also be used, to ensure that the original delivery also proceeds. In this case, all the user has to do is to create a file called, say, *.vacation*, containing a vacation message.

Another advantage of both these methods is that they both work even when the use of arbitrary pipes by users is locked out.

47.10 Taking copies of mail

Some installations have policies that require archive copies of all messages to be made. A single copy of each message can easily be taken by an appropriate command in a system filter, which could, for example, use a different file for each day's messages.

There is also a shadow transport mechanism that can be used to take copies of messages that are successfully delivered by local transports, one copy per delivery. This could be used, *inter alia*, to implement automatic notification of delivery by sites that insist on doing such things.

47.11 Intermittently connected hosts

It has become quite common (because it is cheaper) for hosts to connect to the Internet periodically rather than remain connected all the time. The normal arrangement is that mail for such hosts accumulates on a system that is permanently connected.

Exim was designed for use on permanently connected hosts, and so it is not particularly well-suited to use in an intermittently connected environment. Nevertheless there are some features that can be used.

47.12 Exim on the upstream server host

It is tempting to arrange for incoming mail for the intermittently connected host to remain on Exim's queue until the client connects. However, this approach does not scale very well. Two different kinds of waiting message are being mixed up in the same queue – those that cannot be delivered because of some temporary problem, and those that are waiting for their destination host to connect. This makes it hard to manage the queue, as well as wasting resources, because each queue runner scans the entire queue.

A better approach is to separate off those messages that are waiting for an intermittently connected host. This can be done by delivering these messages into local files in batch SMTP, "mailstore", or other envelope-preserving format, from where they are transmitted by other software when their destination connects. This makes it easy to collect all the mail for one host in a single directory, and to apply local timeout rules on a per-message basis if required.

On a very small scale, leaving the mail on Exim's queue can be made to work. If you are doing this, you should configure Exim with a long retry period for the intermittent host. For example:

```
cheshire.wonderland.fict.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 intermittent host comes online, forcing delivery of one message (either by using the **-M** or **-R** options, or by using the ETRN SMTP command (see section 45.8) causes all the queued up messages to be delivered, often down a single SMTP connection. While the host remains connected, any new messages get 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 get confused, because the IP address is normally used as part of the key string for holding retry information. This can be avoided by unsetting **retry_include_ip_address** on the *smtp* transport. Since this has disadvantages for permanently connected hosts, it is best to arrange a separate transport for the intermittently connected ones.

47.13 Exim on the intermittently connected client host

The value of **smtp_accept_queue_per_connection** should probably be increased, or even set to zero (that is, disabled) on the intermittently connected host, so that all incoming messages down a single connection get delivered immediately.

Mail waiting to be sent from an intermittently connected host will probably not have been routed, because without a connection DNS lookups are not possible. This means that if a normal queue run is done at connection time, each message is likely to be sent in a separate SMTP session. This can be avoided by starting the queue run with a command line option beginning with **-qq** instead of **-q**. In this case, the queue is scanned twice. In the first pass, routing is done but no deliveries take place. The second pass is a normal queue run; since all the messages have been previously routed, those destined for the same host are likely to get sent as multiple deliveries in a single SMTP connection.

48. Using Exim as a non-queueing client

On a personal computer, it is a common requirement for all email to be sent to a "smart host". There are plenty of MUAs that can be configured to operate that way, for all the popular operating systems. However, there are some MUAs for Unix-like systems that cannot be so configured: they submit messages using the command line interface of */usr/sbin/sendmail*. Furthermore, utility programs such as *cron* submit messages this way.

If the personal computer runs continuously, there is no problem, because it can run a conventional MTA that handles delivery to the smart host, and deal with any delays via its queueing mechanism. However, if the computer does not run continuously or runs different operating systems at different times, queueing email is not desirable.

There is therefore a requirement for something that can provide the */usr/sbin/sendmail* interface but deliver messages to a smart host without any queueing or retrying facilities. Furthermore, the delivery to the smart host should be synchronous, so that if it fails, the sending MUA is immediately informed. In other words, we want something that extends an MUA that submits to a local MTA via the command line so that it behaves like one that submits to a remote smart host using TCP/SMTP.

There are a number of applications (for example, there is one called *ssmtp*) that do this job. However, people have found them to be lacking in various ways. For instance, you might want to allow aliasing and forwarding to be done before sending a message to the smart host.

Exim already had the necessary infrastructure for doing this job. Just a few tweaks were needed to make it behave as required, though it is somewhat of an overkill to use a fully-featured MTA for this purpose.

There is a Boolean global option called **mua_wrapper**, defaulting false. Setting **mua_wrapper** true causes Exim to run in a special mode where it assumes that it is being used to "wrap" a command-line MUA in the manner just described. As well as setting **mua_wrapper**, you also need to provide a compatible router and transport configuration. Typically there will be just one router and one transport, sending everything to a smart host.

When run in MUA wrapping mode, the behaviour of Exim changes in the following ways:

- A daemon cannot be run, nor will Exim accept incoming messages from *inetd*. In other words, the only way to submit messages is via the command line.
- Each message is synchronously delivered as soon as it is received (**-odi** is assumed). All queueing options (**queue_only**, **queue_smtp_domains**, **control** in an ACL, etc.) are quietly ignored. The Exim reception process does not finish until the delivery attempt is complete. If the delivery is successful, a zero return code is given.
- Address redirection is permitted, but the final routing for all addresses must be to the same remote transport, and to the same list of hosts. Furthermore, the return address (envelope sender) must be the same for all recipients, as must any added or deleted header lines. In other words, it must be possible to deliver the message in a single SMTP transaction, however many recipients there are.
- If these conditions are not met, or if routing any address results in a failure or defer status, or if Exim is unable to deliver all the recipients successfully to one of the smart hosts, delivery of the entire message fails.
- Because no queueing is allowed, all failures are treated as permanent; there is no distinction between 4*xx* and 5*xx* SMTP response codes from the smart host. Furthermore, because only a single yes/no response can be given to the caller, it is not possible to deliver to some recipients and not others. If there is an error (temporary or permanent) for any recipient, all are failed.
- If more than one smart host is listed, Exim will try another host after a connection failure or a timeout, in the normal way. However, if this kind of failure happens for all the hosts, the delivery fails.

- When delivery fails, an error message is written to the standard error stream (as well as to Exim's log), and Exim exits to the caller with a return code value 1. The message is expunged from Exim's spool files. No bounce messages are ever generated.

- No retry data is maintained, and any retry rules are ignored.

- A number of Exim options are overridden: **deliver_drop_privilege** is forced true, **max_rcpt** in the *smtp* transport is forced to "unlimited", **remote_max_parallel** is forced to one, and fallback hosts are ignored.

The overall effect is that Exim makes a single synchronous attempt to deliver the message, failing if there is any kind of problem. Because no local deliveries are done and no daemon can be run, Exim does not need root privilege. It should be possible to run it setuid to *exim* instead of setuid to *root*. See section 52.3 for a general discussion about the advantages and disadvantages of running without root privilege.

49. Log files

Exim writes three different logs, referred to as the main log, the reject log, and the panic log:

- The main log records the arrival of each message and each delivery in a single 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 recorded in the main log. Some of them are optional, in which case the **log_selector** option controls whether they are included or not. A Perl script called *eximstats*, which does simple analysis of main log files, is provided in the Exim distribution (see section 50.7).
- The reject log records information from messages that are rejected as a result of a configuration option (that is, for policy reasons). The first line of each rejection is a copy of the line that is also written to the main log. Then, if the message's header has been read at the time the log is written, its contents are written to this log. Only the original header lines are available; header lines added by ACLs are not logged. You can use the reject log to check that your policy controls are working correctly; on a busy host this may be easier than scanning the main log for rejection messages. You can suppress the writing of the reject log by setting **write_rejectlog** false.
- When certain serious errors occur, Exim writes entries to its panic log. If the error is sufficiently disastrous, Exim bombs out afterwards. Panic log entries are usually written to the main log as well, but can get lost amid the mass of other entries. The panic log should be empty under normal circumstances. It is therefore a good idea to check it (or to have a *cron* script check it) regularly, in order to become aware of any problems. When Exim cannot open its panic log, it tries as a last resort to write to the system log (syslog). This is opened with LOG_PID+LOG_CONS and the facility code of LOG_MAIL. The message itself is written at priority LOG_CRIT.

Every log line starts with a timestamp, in the format shown in the following example. Note that many of the examples shown in this chapter are line-wrapped. In the log file, this would be all on one line:

```
2001-09-16 16:09:47 SMTP connection from [127.0.0.1] closed
  by QUIT
```

By default, the timestamps are in the local timezone. There are two ways of changing this:

- You can set the **timezone** option to a different time zone; in particular, if you set

  ```
  timezone = UTC
  ```

  the timestamps will be in UTC (aka GMT).
- If you set **log_timezone** true, the time zone is added to the timestamp, for example:

  ```
  2003-04-25 11:17:07 +0100 Start queue run: pid=12762
  ```

Exim does not include its process id in log lines by default, but you can request that it does so by specifying the `pid` log selector (see section 49.15). When this is set, the process id is output, in square brackets, immediately after the time and date.

49.1 Where the logs are written

The logs may be written to local files, or to syslog, or both. However, it should be noted that many syslog implementations use UDP as a transport, and are therefore unreliable in the sense that messages are not guaranteed to arrive at the loghost, nor is the ordering of messages necessarily maintained. It has also been reported that on large log files (tens of megabytes) you may need to tweak syslog to prevent it syncing the file with each write – on Linux this has been seen to make syslog take 90% plus of CPU time.

The destination for Exim's logs is configured by setting LOG_FILE_PATH in *Local/Makefile* or by setting **log_file_path** in the run time configuration. This latter string is expanded, so it can contain, for example, references to the host name:

```
log_file_path = /var/log/$primary_hostname/exim_%slog
```

It is generally advisable, however, to set the string in *Local/Makefile* rather than at run time, because then the setting is available right from the start of Exim's execution. Otherwise, if there's 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 or **log_file_path** is a colon-separated list, currently limited to at most two items. This is one option where the facility for changing a list separator may not be used. The list must always be colon-separated. If an item in the list 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 a default path.

When Exim encounters an empty item in the list, it searches the list defined by LOG_FILE_PATH, and uses the first item it finds that is neither empty nor "syslog". This means that an empty item in **log_file_path** can be used to mean "use the path specified at build time". It no such item exists, log files are written in the *log* subdirectory of the spool directory. This is equivalent to the setting:

```
log_file_path = $spool_directory/log/%slog
```

If you do not specify anything at build time or run time, that is where the logs are written.

A log file path may also contain `%D` if datestamped log file names are in use – see section 49.3 below.

Here are some examples of possible settings:

`LOG_FILE_PATH=syslog`	syslog only
`LOG_FILE_PATH=:syslog`	syslog and default path
`LOG_FILE_PATH=syslog : /usr/log/exim_%s`	syslog and specified path
`LOG_FILE_PATH=/usr/log/exim_%s`	specified path only

If there are more than two paths in the list, the first is used and a panic error is logged.

49.2 Logging to local files that are periodically "cycled"

Some operating systems provide centralized and standardized methods for cycling log files. For those that do not, a utility script called *exicyclog* is provided (see section 50.6). This renames and compresses the main and reject logs each time it is called. The maximum number of old logs to keep can be set. It is suggested this script is run as a daily *cron* job.

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 may be kept open long after it is renamed if *exicyclog* or something similar is being used to rename log files on a regular basis. To ensure that a switch of log files is noticed as soon as possible, Exim calls *stat()* on the main log's name before reusing an open file, and if the file does not exist, or its inode has changed, the old file is closed and Exim tries to open the main log 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.

49.3 Datestamped log files

Instead of cycling the main and reject log files by renaming them periodically, some sites like to use files whose names contain a datestamp, for example, *mainlog-20031225*. The datestamp is in the form *yyyymmdd*. Exim has support for this way of working. It is enabled by setting the **log_file_path** option to a path that includes `%D` at the point where the datestamp is required. For example:

```
log_file_path = /var/spool/exim/log/%slog-%D
log_file_path = /var/log/exim-%s-%D.log
log_file_path = /var/spool/exim/log/%D-%slog
```

As before, `%s` is replaced by "main" or "reject"; the following are examples of names generated by the above examples:

```
/var/spool/exim/log/mainlog-20021225
/var/log/exim-reject-20021225.log
/var/spool/exim/log/20021225-mainlog
```

When this form of log file is specified, Exim automatically switches to new files at midnight. It does not make any attempt to compress old logs; you will need to write your own script if you require this. You should not run *exicyclog* with this form of logging.

The location of the panic log is also determined by **log_file_path**, but it is not datestamped, because rotation of the panic log does not make sense. When generating the name of the panic log, `%D` is removed from the string. In addition, if it immediately follows a slash, a following non-alphanumeric character is removed; otherwise a preceding non-alphanumeric character is removed. Thus, the three examples above would give these panic log names:

```
/var/spool/exim/log/paniclog
/var/log/exim-panic.log
/var/spool/exim/log/paniclog
```

49.4 Logging to syslog

The use of syslog does not change what Exim logs or the format of its messages, except in one respect. If **syslog_timestamp** is set false, the timestamps on Exim's log lines are omitted when these lines are sent to syslog. Apart from that, 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" by default, but you can change these by setting the **syslog_facility** and **syslog_processname** options, respectively. If Exim was compiled with SYSLOG_LOG_PID set in *Local/Makefile* (this is the default in *src/EDITME*), then, 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 host name to each line. The three log streams are mapped onto syslog priorities as follows:

- *mainlog* is mapped to LOG_INFO
- *rejectlog* is mapped to LOG_NOTICE
- *paniclog* is mapped to LOG_ALERT

Many log lines are written to both *mainlog* and *rejectlog*, and some are written to both *mainlog* and *paniclog*, so there will be duplicates if these are routed by syslog to the same place. You can suppress this duplication by setting **syslog_duplication** false.

Exim's log lines can sometimes be very long, and some of its *rejectlog* 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 870 data characters. (This allows for a total syslog line length of 1024, when additions such as timestamps are added.) If you are running a syslog replacement that can handle lines longer than the 1024 characters allowed by RFC 3164, you should set

```
SYSLOG_LONG_LINES=yes
```

in *Local/Makefile* before building Exim. That stops Exim from splitting long lines, but it still splits at internal newlines in *reject* log entries.

To make it easy to re-assemble split lines 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 / delimiter is used when the line was split because it was too long; if it was split because of an internal newline, the \ delimiter is used. For example, supposing the length limit to be 50 instead of 870, the following would be the result of a typical rejection message to *mainlog* (LOG_INFO), each line in addition being preceded by the time, host name, and pid as added by syslog:

```
[1/5] 2002-09-16 16:09:43 16RdAL-0006pc-00 rejected from
[2/5]  [127.0.0.1] (ph10): syntax error in 'From' header
[3/5]  when scanning for sender: missing or malformed lo
```

```
[4/5] cal part in "<>" (envelope sender is <ph10@cam.exa
[5/5] mple>)
```

The same error might cause the following lines to be written to "rejectlog" (LOG_NOTICE):

```
[1/18] 2002-09-16 16:09:43 16RdAL-0006pc-00 rejected fro
[2/18] m [127.0.0.1] (ph10): syntax error in 'From' head
[3/18] er when scanning for sender: missing or malformed
[4/18]  local part in "<>" (envelope sender is <ph10@cam
[5\18] .example>)
[6\18] Recipients: ph10@some.domain.cam.example
[7\18] P Received: from [127.0.0.1] (ident=ph10)
[8\18]        by xxxxx.cam.example with smtp (Exim 4.00)
[9\18]        id 16RdAL-0006pc-00
[10/18]        for ph10@cam.example; Mon, 16 Sep 2002 16:
[11\18] 09:43 +0100
[12\18] F From: <>
[13\18]   Subject: this is a test header
[18\18]   X-something: this is another header
[15/18] I Message-Id: <E16RdAL-0006pc-00@xxxxx.cam.examp
[16\18] le>
[17\18] B Bcc:
[18/18]   Date: Mon, 16 Sep 2002 16:09:43 +0100
```

Log lines that are neither too long nor contain newlines are written to syslog without modification.

If only syslog is being used, the Exim monitor is unable to provide a log tail display, unless syslog is routing *mainlog* to a file on the local host and the environment variable EXIMON_LOG_FILE_PATH is set to tell the monitor where it is.

49.5 Log line flags

One line is written to the main log for each message received, and for each successful, unsuccessful, and delayed delivery. These lines can readily be picked out by the distinctive two-character flags that immediately follow the timestamp. The flags are:

<=	message arrival
=>	normal message delivery
->	additional address in same delivery
*>	delivery suppressed by **-N**
**	delivery failed; address bounced
==	delivery deferred; temporary problem

49.6 Logging message reception

The format of the single-line entry in the main log that is written for every message received is shown in the basic example below, which is split over several lines in order to fit it on the page:

```
2002-10-31 08:57:53 16ZCW1-0005MB-00 <= kryten@dwarf.fict.example
  H=mailer.fict.example [192.168.123.123] U=exim
  P=smtp S=5678 id=<incoming message id>
```

The address immediately following "<=" is the envelope sender address. A bounce message is shown with the sender address "<>", and if it is locally generated, this is followed by an item of the form

```
R=<message id>
```

which is a reference to the message that caused the bounce to be sent.

For messages from other hosts, the H and U fields identify the remote host and record the RFC 1413 identity of the user that sent the message, if one was received. The number given in square brackets is the IP address of the sending host. If there is a single, unparenthesized host name in the H field, as above, it has been verified to correspond to the IP address (see the **host_lookup** option). If the name

is in parentheses, it was the name quoted by the remote host in the SMTP HELO or EHLO command, and has not been verified. If verification yields a different name to that given for HELO or EHLO, the verified name appears first, followed by the HELO or EHLO name in parentheses.

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 text like these examples:

```
H=(10.21.32.43) [192.168.8.34]
H=([10.21.32.43]) [192.168.8.34]
```

This can be confusing. Only the final address in square brackets can be relied on.

For locally generated messages (that is, messages not received over TCP/IP), the H field is omitted, and the U field contains the login name of the caller of Exim.

For all messages, the P field specifies the protocol used to receive the message. This is the value that is stored in *$received_protocol*. In the case of incoming SMTP messages, the value indicates whether or not any SMTP extensions (ESMTP), encryption, or authentication were used. If the SMTP session was encrypted, there is an additional X field that records the cipher suite that was used.

The protocol is set to "esmtpsa" or "esmtpa" for messages received from hosts that have authenticated themselves using the SMTP AUTH command. The first value is used when the SMTP connection was encrypted ("secure"). In this case there is an additional item A= followed by the name of the authenticator that was used. 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.

The id field records the existing message id, if present. The size of the received message is given 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 not correspond with this value (and indeed may be different to each other).

The **log_selector** option can be used to request the logging of additional data when a message is received. See section 49.15 below.

49.7 Logging deliveries

The format of the single-line entry in the main log that is written for every delivery is shown in one of the examples below, for local and remote deliveries, respectively. Each example has been split into two lines in order to fit it on the page:

```
2002-10-31 08:59:13 16ZCW1-0005MB-00 => marv
  <marv@hitch.fict.example> R=localuser T=local_delivery
2002-10-31 09:00:10 16ZCW1-0005MB-00 =>
  monk@holistic.fict.example R=dnslookup T=remote_smtp
  H=holistic.fict.example [192.168.234.234]
```

For ordinary local deliveries, the original address is given in angle brackets after the final delivery address, which might be a pipe or a file. If intermediate address(es) exist between the original and the final address, the last of these is given in parentheses after the final address. The R and T fields record the router and transport that were used to process the address.

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.

When more than one address is included in a single delivery (for example, two SMTP RCPT commands in one transaction) the second and subsequent addresses are flagged with -> instead of =>. When 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.

The generation of a reply message by a filter file gets logged as a "delivery" to the addressee, preceded by ">".

The **log_selector** option can be used to request the logging of additional data when a message is delivered. See section 49.15 below.

49.8 Discarded deliveries

When a message is discarded as a result of the command "seen finish" being obeyed in a filter file which generates no deliveries, a log entry of the form

```
2002-12-10 00:50:49 16auJc-0001UB-00 => discarded
  <low.club@bridge.example> R=userforward
```

is written, to record why no deliveries are logged. When a message is discarded because it is aliased to ":blackhole:" the log line is like this:

```
1999-03-02 09:44:33 10HmaX-0005vi-00 => :blackhole:
  <hole@nowhere.example> R=blackhole_router
```

49.9 Deferred deliveries

When a delivery is deferred, a line of the following form is logged:

```
2002-12-19 16:20:23 16aiQz-0002Q5-00 == marvin@endrest.example
  R=dnslookup T=smtp defer (146): Connection refused
```

In the case of remote deliveries, the error is the one that was given for the last IP address that was tried. Details of individual SMTP failures are also written to the log, so the above line would be preceded by something like

```
2002-12-19 16:20:23 16aiQz-0002Q5-00 Failed to connect to
  mail1.endrest.example [192.168.239.239]: Connection refused
```

When a deferred address is skipped because its retry time has not been reached, a message is written to the log, but this can be suppressed by setting an appropriate value in **log_selector**.

49.10 Delivery failures

If a delivery fails because an address cannot be routed, 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
```

If a delivery fails at transport time, the router and transport are shown, and the response from the remote host is included, as in this example:

```
2002-07-11 07:14:17 17SXDU-000189-00 ** ace400@pb.example
  R=dnslookup T=remote_smtp: SMTP error from remote mailer
  after pipelined RCPT TO:<ace400@pb.example>: host
  pbmail3.py.example [192.168.63.111]: 553 5.3.0
  <ace400@pb.example>...Addressee unknown
```

The word "pipelined" indicates that the SMTP PIPELINING extension was being used. See **hosts_avoid_esmtp** in the *smtp* transport for a way of disabling PIPELINING. The log lines for all forms of delivery failure are flagged with `**`.

49.11 Fake deliveries

If a delivery does not actually take place because the **-N** option has been used to suppress it, a normal delivery line is written to the log, except that "=>" is replaced by "*>".

49.12 Completion

A line of the form

```
2002-10-31 09:00:11 16ZCW1-0005MB-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.

49.13 Summary of Fields in Log Lines

A summary of the field identifiers that are used in log lines is shown in the following table:

`A`	authenticator name (and optional id)
`C`	SMTP confirmation on delivery
	command list for "no mail in SMTP session"
`CV`	certificate verification status
`D`	duration of "no mail in SMTP session"
`DN`	distinguished name from peer certificate
`DT`	on => lines: time taken for a delivery
`F`	sender address (on delivery lines)
`H`	host name and IP address
`I`	local interface used
`id`	message id for incoming message
`P`	on <= lines: protocol used
	on => and `**` lines: return path
`QT`	on => lines: time spent on queue so far
	on "Completed" lines: time spent on queue
`R`	on <= lines: reference for local bounce
	on => `**` and == lines: router name
`S`	size of message
`ST`	shadow transport name
`T`	on <= lines: message subject (topic)
	on => `**` and == lines: transport name
`U`	local user or RFC 1413 identity
`X`	TLS cipher suite

49.14 Other log entries

Various other types of log entry are written from time to time. Most should be self-explanatory. Among the more common are:

- *retry time not reached* An address previously suffered a temporary error during routing or local delivery, and the time to retry has not yet arrived. This message is not written to an individual message log file unless it happens during the first delivery attempt.
- *retry time not reached for any host* An 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.
- *spool file locked* An attempt to deliver a message cannot proceed because some other Exim process is already working on the message. This can be quite common if queue running processes are started at frequent intervals. The *exiwhat* utility script can be used to find out what Exim processes are doing.
- *error ignored* There are several circumstances that give rise to this message:

  (1) Exim failed to deliver a bounce message whose age was greater than **ignore_bounce_errors_after**. The bounce was discarded.

  (2) A filter file set up a delivery using the "noerror" option, and the delivery failed. The delivery was discarded.

  (3) A delivery set up by a router configured with

  ```
  errors_to = <>
  ```

  failed. The delivery was discarded.

49.15 Reducing or increasing what is logged

By setting the **log_selector** global option, you can disable some of Exim's default logging, or you can request additional logging. The value of **log_selector** is made up of names preceded by plus or minus characters. For example:

```
log_selector = +arguments -retry_defer
```

The list of optional log items is in the following table, with the default selection marked by asterisks:

*acl_warn_skipped	skipped **warn** statement in ACL
address_rewrite	address rewriting
all_parents	all parents in => lines
arguments	command line arguments
*connection_reject	connection rejections
*delay_delivery	immediate delivery delayed
deliver_time	time taken to perform delivery
delivery_size	add S=*nnn* to => lines
*dnslist_defer	defers of DNS list (aka RBL) lookups
*etrn	ETRN commands
*host_lookup_failed	as it says
ident_timeout	timeout for ident connection
incoming_interface	incoming interface on <= lines
incoming_port	incoming port on <= lines
*lost_incoming_connection	as it says (includes timeouts)
outgoing_port	add remote port to => lines
*queue_run	start and end queue runs
queue_time	time on queue for one recipient
queue_time_overall	time on queue for whole message
pid	Exim process id
received_recipients	recipients on <= lines
received_sender	sender on <= lines
*rejected_header	header contents on reject log
*retry_defer	"retry time not reached"
return_path_on_delivery	put return path on => and ** lines
sender_on_delivery	add sender to => lines
*sender_verify_fail	sender verification failures
*size_reject	rejection because too big
*skip_delivery	delivery skipped in a queue run
smtp_confirmation	SMTP confirmation on => lines
smtp_connection	SMTP connections
smtp_incomplete_transaction	incomplete SMTP transactions
smtp_no_mail	session with no MAIL commands
smtp_protocol_error	SMTP protocol errors
smtp_syntax_error	SMTP syntax errors
subject	contents of *Subject:* on <= lines
tls_certificate_verified	certificate verification status
*tls_cipher	TLS cipher suite on <= and => lines
tls_peerdn	TLS peer DN on <= and => lines
unknown_in_list	DNS lookup failed in list match
all	all of the above

More details on each of these items follows:

- **acl_warn_skipped**: When an ACL **warn** statement is skipped because one of its conditions cannot be evaluated, a log line to this effect is written if this log selector is set.
- **address_rewrite**: This applies both to global rewrites and per-transport rewrites, but not to rewrites in filters run as an unprivileged user (because such users cannot access the log).

- **all_parents**: Normally only the original and final addresses are logged on delivery lines; with this selector, intermediate parents are given in parentheses between them.
- **arguments**: This causes Exim to write the arguments with which it was called to the main log, preceded by the current working directory. This is a debugging feature, added to make it easier to find out how certain MUAs call */usr/sbin/sendmail*. The logging does not happen if Exim has given up root privilege because it was called with the **-C** or **-D** options. Arguments that are empty or that contain white space are quoted. Non-printing characters are shown as escape sequences. This facility cannot log unrecognized arguments, because the arguments are checked before the configuration file is read. The only way to log such cases is to interpose a script such as *util/logargs.sh* between the caller and Exim.
- **connection_reject**: A log entry is written whenever an incoming SMTP connection is rejected, for whatever reason.
- **delay_delivery**: A log entry is written whenever a delivery process is not started for an incoming message because the load is too high or too many messages were received on one connection. Logging does not occur if no delivery process is started because **queue_only** is set or **-odq** was used.
- **deliver_time**: For each delivery, the amount of real time it has taken to perform the actual delivery is logged as DT=<*time*>, for example, `DT=1s`.
- **delivery_size**: For each delivery, the size of message delivered is added to the "=>" line, tagged with S=.
- **dnslist_defer**: A log entry is written if an attempt to look up a host in a DNS black list suffers a temporary error.
- **etrn**: Every valid ETRN command that is received is logged, before the ACL is run to determine whether or not it is actually accepted. An invalid ETRN command, or one received within a message transaction is not logged by this selector (see **smtp_syntax_error** and **smtp_protocol_error**).
- **host_lookup_failed**: When a lookup of a host's IP addresses fails to find any addresses, or when a lookup of an IP address fails to find a host name, a log line is written. This logging does not apply to direct DNS lookups when routing email addresses, but it does apply to "byname" lookups.
- **ident_timeout**: A log line is written whenever an attempt to connect to a client's ident port times out.
- **incoming_interface**: The interface on which a message was received is added to the "<=" line as an IP address in square brackets, tagged by I= and followed by a colon and the port number. The local interface and port are also added to other SMTP log lines, for example "SMTP connection from", and to rejection lines.
- **incoming_port**: The remote port number from which a message was received is added to log entries and *Received:* header lines, following the IP address in square brackets, and separated from it by a colon. This is implemented by changing the value that is put in the *$sender_fullhost* and *$sender_rcvhost* variables. Recording the remote port number has become more important with the widening use of NAT (see RFC 2505).
- **lost_incoming_connection**: A log line is written when an incoming SMTP connection is unexpectedly dropped.
- **outgoing_port**: The remote port number is added to delivery log lines (those containing => tags) following the IP address. This option is not included in the default setting, because for most ordinary configurations, the remote port number is always 25 (the SMTP port).
- **pid**: The current process id is added to every log line, in square brackets, immediately after the time and date.
- **queue_run**: The start and end of every queue run are logged.
- **queue_time**: The amount of time the message has been in the queue on the local host is logged as QT=<*time*> on delivery (=>) lines, for example, `QT=3m45s`. The clock starts when Exim starts to

receive the message, so it includes reception time as well as the delivery time for the current address. This means that it may be longer than the difference between the arrival and delivery log line times, because the arrival log line is not written until the message has been successfully received.

- **queue_time_overall**: The amount of time the message has been in the queue on the local host is logged as QT=<*time*> on "Completed" lines, for example, `QT=3m45s`. The clock starts when Exim starts to receive the message, so it includes reception time as well as the total delivery time.

- **received_recipients**: The recipients of a message are listed in the main log as soon as the message is received. The list appears at the end of the log line that is written when a message is received, preceded by the word "for". The addresses are listed after they have been qualified, but before any rewriting has taken place. Recipients that were discarded by an ACL for MAIL or RCPT do not appear in the list.

- **received_sender**: The unrewritten original sender of a message is added to the end of the log line that records the message's arrival, after the word "from" (before the recipients if **received_recipients** is also set).

- **rejected_header**: If a message's header has been received at the time a rejection is written to the reject log, the complete header is added to the log. Header logging can be turned off individually for messages that are rejected by the *local_scan()* function (see section 42.2).

- **retry_defer**: A log line is written if a delivery is deferred because a retry time has not yet been reached. However, this "retry time not reached" message is always omitted from individual message logs after the first delivery attempt.

- **return_path_on_delivery**: The return path that is being transmitted with the message is included in delivery and bounce lines, using the tag P=. This is omitted if no delivery actually happens, for example, if routing fails, or if delivery is to */dev/null* or to `:blackhole:`.

- **sender_on_delivery**: The message's sender address is added to every delivery and bounce line, tagged by F= (for "from"). This is the original sender that was received with the message; it is not necessarily the same as the outgoing return path.

- **sender_verify_fail**: If this selector is unset, the separate log line that gives details of a sender verification failure is not written. Log lines for the rejection of SMTP commands contain just "sender verify failed", so some detail is lost.

- **size_reject**: A log line is written whenever a message is rejected because it is too big.

- **skip_delivery**: A log line is written whenever a message is skipped during a queue run because it is frozen or because another process is already delivering it. The message that is written is "spool file is locked".

- **smtp_confirmation**: The response to the final "." in the SMTP dialogue for outgoing messages is added to delivery log lines in the form `C=`<*text*>. A number of MTAs (including Exim) return an identifying string in this response.

- **smtp_connection**: A log line is written whenever an SMTP connection is established or closed, unless the connection is from a host that matches **hosts_connection_nolog**. (In contrast, **lost_incoming_connection** applies only when the closure is unexpected.) This applies to connections from local processes that use **-bs** as well as to TCP/IP connections. If a connection is dropped in the middle of a message, a log line is always written, whether or not this selector is set, but otherwise nothing is written at the start and end of connections unless this selector is enabled.

  For TCP/IP connections to an Exim daemon, the current number of connections is included in the log message for each new connection, but note that the count is reset if the daemon is restarted. Also, because connections are closed (and the closure is logged) in subprocesses, the count may not include connections that have been closed but whose termination the daemon has not yet noticed. Thus, while it is possible to match up the opening and closing of connections in the log, the value of the logged counts may not be entirely accurate.

- **smtp_incomplete_transaction**: When a mail transaction is aborted by RSET, QUIT, loss of connection, or otherwise, the incident is logged, and the message sender plus any accepted recipients are included in the log line. This can provide evidence of dictionary attacks.

- **smtp_no_mail**: A line is written to the main log whenever an accepted SMTP connection terminates without having issued a MAIL command. This includes both the case when the connection is dropped, and the case when QUIT is used. It does not include cases where the connection is rejected right at the start (by an ACL, or because there are too many connections, or whatever). These cases already have their own log lines.

  The log line that is written contains the identity of the client in the usual way, followed by D= and a time, which records the duration of the connection. If the connection was authenticated, this fact is logged exactly as it is for an incoming message, with an A= item. If the connection was encrypted, CV=, DN=, and X= items may appear as they do for an incoming message, controlled by the same logging options.

  Finally, if any SMTP commands were issued during the connection, a C= item is added to the line, listing the commands that were used. For example,

  ```
  C=EHLO,QUIT
  ```

  shows that the client issued QUIT straight after EHLO. If there were fewer than 20 commands, they are all listed. If there were more than 20 commands, the last 20 are listed, preceded by "...". However, with the default setting of 10 for **smtp_accep_max_nonmail**, the connection will in any case have been aborted before 20 non-mail commands are processed.

- **smtp_protocol_error**: A log line is written for every SMTP protocol error encountered. Exim does not have perfect detection of all protocol errors because of transmission delays and the use of pipelining. If PIPELINING has been advertised to a client, an Exim server assumes that the client will use it, and therefore it does not count "expected" errors (for example, RCPT received after rejecting MAIL) as protocol errors.

- **smtp_syntax_error**: A log line is written for every SMTP syntax error encountered. 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.

- **subject**: The subject of the message is added to the arrival log line, preceded by "T=" (T for "topic", since S is already used for "size"). Any MIME "words" in the subject are decoded. The **print_topbitchars** option specifies whether characters with values greater than 127 should be logged unchanged, or whether they should be rendered as escape sequences.

- **tls_certificate_verified**: An extra item is added to <= and => log lines when TLS is in use. The item is `CV=yes` if the peer's certificate was verified, and `CV=no` if not.

- **tls_cipher**: When a message is sent or received over an encrypted connection, the cipher suite used is added to the log line, preceded by X=.

- **tls_peerdn**: When a message is sent or received over an encrypted connection, and a certificate is supplied by the remote host, the peer DN is added to the log line, preceded by DN=.

- **unknown_in_list**: This setting causes a log entry to be written when the result of a list match is failure because a DNS lookup failed.

49.16 Message log

In addition to the general log files, Exim writes a log file for each message that it handles. The names of these per-message logs are the message ids, and they are kept in the *msglog* sub-directory of the spool directory. Each message log contains copies of the log lines that apply to the message. This makes it easier to inspect the status of an individual message without having to search the main log. A message log is deleted when processing of the message is complete, unless **preserve_message_logs** is set, but this should be used only with great care because they can fill up your disk very quickly.

On a heavily loaded system, it may be desirable to disable the use of per-message logs, in order to reduce disk I/O. This can be done by setting the **message_logs** option false.

50. Exim utilities

A number of utility scripts and programs are supplied with Exim and are described in this chapter. There is also the Exim Monitor, which is covered in the next chapter. The utilities described here are:

50.1	*exiwhat*	list what Exim processes are doing
50.2	*exiqgrep*	grep the queue
50.3	*exiqsumm*	summarize the queue
50.4	*exigrep*	search the main log
50.5	*exipick*	select messages on various criteria
50.6	*exicyclog*	cycle (rotate) log files
50.7	*eximstats*	extract statistics from the log
50.8	*exim_checkaccess*	check address acceptance from given IP
50.9	*exim_dbmbuild*	build a DBM file
50.10	*exinext*	extract retry information
50.11	*exim_dumpdb*	dump a hints database
50.11	*exim_tidydb*	clean up a hints database
50.11	*exim_fixdb*	patch a hints database
50.15	*exim_lock*	lock a mailbox file

Another utility that might be of use to sites with many MTAs is Tom Kistner's *exilog*. It provides log visualizations across multiple Exim servers. See **http://duncanthrax.net/exilog/** for details.

50.1 Finding out what Exim processes are doing (exiwhat)

On operating systems that can restart a system call after receiving a signal (most modern OS), an Exim process responds to the SIGUSR1 signal by writing a line describing what it is doing to the file *exim-process.info* in the Exim spool directory. The *exiwhat* script sends the signal to all Exim processes it can find, having first emptied the file. It then waits for one second to allow the Exim processes to react before displaying the results. In order to run *exiwhat* successfully you have to have sufficient privilege to send the signal to the Exim processes, so it is normally run as root.

Warning: This is not an efficient process. It is intended for occasional use by system administrators. It is not sensible, for example, to set up a script that sends SIGUSR1 signals to Exim processes at short intervals.

Unfortunately, the *ps* command that *exiwhat* uses to find Exim processes varies in different operating systems. Not only are different options used, but the format of the output is different. For this reason, there are some system configuration options that configure exactly how *exiwhat* works. If it doesn't seem to be working for you, check the following compile-time options:

`EXIWHAT_PS_CMD`	the command for running *ps*
`EXIWHAT_PS_ARG`	the argument for *ps*
`EXIWHAT_EGREP_ARG`	the argument for *egrep* to select from *ps* output
`EXIWHAT_KILL_ARG`	the argument for the *kill* command

An example of typical output from *exiwhat* is

```
164 daemon: -q1h, listening on port 25
10483 running queue: waiting for 0tAycK-0002ij-00 (10492)
10492 delivering 0tAycK-0002ij-00 to mail.ref.example
  [10.19.42.42] (editor@ref.example)
10592 handling incoming call from [192.168.243.242]
10628 accepting a local non-SMTP message
```

The first number in the output line is the process number. The third line has been split here, in order to fit it on the page.

50.2 Selective queue listing (exiqgrep)

This utility is a Perl script contributed by Matt Hubbard. It runs

```
exim -bpu
```

to obtain a queue listing with undelivered recipients only, and then greps the output to select messages that match given criteria. The following selection options are available:

-f <*regex*>
Match the sender address. The field that is tested is enclosed in angle brackets, so you can test for bounce messages with

```
exiqgrep -f '^<>$'
```

-r <*regex*>
Match a recipient address. The field that is tested is not enclosed in angle brackets.

-s <*regex*>
Match against the size field.

-y <*seconds*>
Match messages that are younger than the given time.

-o <*seconds*>
Match messages that are older than the given time.

-z
Match only frozen messages.

-x
Match only non-frozen messages.

The following options control the format of the output:

-c
Display only the count of matching messages.

-l
Long format – display the full message information as output by Exim. This is the default.

-i
Display message ids only.

-b
Brief format – one line per message.

-R
Display messages in reverse order.

There is one more option, **-h**, which outputs a list of options.

50.3 Summarizing the queue (exiqsumm)

The *exiqsumm* utility is a Perl script which reads the output of `exim -bp` and produces a summary of the messages on the queue. Thus, you use it by running a command such as

```
exim -bp | exiqsumm
```

The output consists of one line for each domain that has messages waiting for it, as in the following example:

```
3   2322   74m   66m  msn.com.example
```

Each line lists the number of pending deliveries for a domain, their total volume, and the length of time that the oldest and the newest messages have been waiting. Note that the number of pending deliveries is greater than the number of messages when messages have more than one recipient.

A summary line is output at the end. By default the output is sorted on the domain name, but *exiqsumm* has the options **-a** and **-c**, which cause the output to be sorted by oldest message and by count of messages, respectively. There are also three options that split the messages for each domain

into two or more subcounts: **-b** separates bounce messages, **-f** separates frozen messages, and **-s** separates messages according to their sender.

The output of *exim -bp* contains the original addresses in the message, so this also applies to the output from *exiqsumm*. No domains from addresses generated by aliasing or forwarding are included (unless the **one_time** option of the *redirect* router has been used to convert them into "top level" addresses).

50.4 Extracting specific information from the log (exigrep)

The *exigrep* utility is a Perl script that searches one or more main log files for entries that match a given pattern. When it finds a match, it extracts all the log entries for the relevant message, not just those that match the pattern. Thus, *exigrep* can extract complete log entries for a given message, or all mail for a given user, or for a given host, for example. The input files can be in Exim log format or syslog format. If a matching log line is not associated with a specific message, it is included in *exigrep*'s output without any additional lines. The usage is:

```
exigrep [-t<n>] [-I] [-l] [-v] <pattern> [<log file>] ...
```

If no log file names are given on the command line, the standard input is read.

The **-t** argument specifies a number of seconds. It adds an additional condition for message selection. Messages that are complete are shown only if they spent more than <*n*> seconds on the queue.

By default, *exigrep* does case-insensitive matching. The **-I** option makes it case-sensitive. This may give a performance improvement when searching large log files. Without **-I**, the Perl pattern matches use Perl's `/i` option; with **-I** they do not. In both cases it is possible to change the case sensitivity within the pattern by using `(?i)` or `(?-i)`.

The **-l** option means "literal", that is, treat all characters in the pattern as standing for themselves. Otherwise the pattern must be a Perl regular expression.

The **-v** option inverts the matching condition. That is, a line is selected if it does *not* match the pattern.

If the location of a *zcat* command is known from the definition of ZCAT_COMMAND in *Local/Makefile*, *exigrep* automatically passes any file whose name ends in COMPRESS_SUFFIX through *zcat* as it searches it.

50.5 Selecting messages by various criteria (exipick)

John Jetmore's *exipick* utility is included in the Exim distribution. It lists messages from the queue according to a variety of criteria. For details of *exipick*'s facilities, visit the web page at **http://www.exim.org/eximwiki/ToolExipickManPage** or run *exipick* with the **--help** option.

50.6 Cycling log files (exicyclog)

The *exicyclog* script can be used to cycle (rotate) *mainlog* and *rejectlog* files. This is not necessary if only syslog is being used, or if you are using log files with datestamps in their names (see section 49.3). Some operating systems have their own standard mechanisms for log cycling, and these can be used instead of *exicyclog* if preferred. There are two command line options for *exicyclog*:

- **-k** <*count*> specifies the number of log files to keep, overriding the default that is set when Exim is built. The default default is 10.
- **-l** <*path*> specifies the log file path, in the same format as Exim's **log_file_path** option (for example, `/var/log/exim_%slog`), again overriding the script's default, which is to find the setting from Exim's configuration.

Each time *exicyclog* is run the file names get "shuffled down" by one. If the main log file name is *mainlog* (the default) then when *exicyclog* is run *mainlog* becomes *mainlog.01*, the previous *mainlog.01* becomes *mainlog.02* and so on, up to the limit that is set in the script or by the **-k** option. Log files whose numbers exceed the limit are discarded. Reject logs are handled similarly.

If the limit is greater than 99, the script uses 3-digit numbers such as *mainlog.001*, *mainlog.002*, etc. If you change from a number less than 99 to one that is greater, or *vice versa*, you will have to fix the names of any existing log files.

If no *mainlog* file exists, the script does nothing. Files that "drop off" the end are deleted. All files with numbers greater than 01 are compressed, using a compression command which is configured by the COMPRESS_COMMAND setting in *Local/Makefile*. It is usual to run *exicyclog* daily from a root **crontab** entry of the form

```
1 0 * * * su exim -c /usr/exim/bin/exicyclog
```

assuming you have used the name "exim" for the Exim user. You can run *exicyclog* as root if you wish, but there is no need.

50.7 Mail statistics (eximstats)

A Perl script called *eximstats* is provided for extracting statistical information from log files. The output is either plain text, or HTML. Exim log files are also supported by the *Lire* system produced by the LogReport Foundation **http://www.logreport.org**.

The *eximstats* script has been hacked about quite a bit over time. The latest version is the result of some extensive revision by Steve Campbell. A lot of information is given 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 /var/spool/exim/log/mainlog.01
```

By default, *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 fifty hosts in each category are listed on the standard output. Similar information, based on email addresses or domains instead of hosts can be requested by means of various options. For messages delivered and received locally, similar statistics are also produced per user.

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 by *eximstats*.

Though normally more deliveries than receipts are reported (as messages may have multiple recipients), it is possible for *eximstats* to report more messages received than delivered, even though the queue 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. A bounce message is handled as an entirely separate message.

eximstats always 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.

The remainder of the output is in sections that can be independently disabled or modified by various options. It consists of a summary of deliveries by transport, histograms of messages received and delivered per time interval (default per hour), information about the time messages spent on the queue, a list of relayed messages, lists of the top fifty sending hosts, local senders, destination hosts, and destination local users by count and by volume, and a list of delivery errors that occurred.

The relay information lists messages that were actually relayed, that is, they came from a remote host and were directly delivered to some other remote host, without being processed (for example, for aliasing or forwarding) locally.

There are quite a few options for *eximstats* to control exactly what it outputs. These are documented in the Perl script itself, and can be extracted by running the command *perldoc* on the script. For example:

```
perldoc /usr/exim/bin/eximstats
```

50.8 Checking access policy (exim_checkaccess)

The **-bh** command line argument allows you to run a fake SMTP session with debugging output, in order to check what Exim is doing when it is applying policy controls to incoming SMTP mail. However, not everybody is sufficiently familiar with the SMTP protocol to be able to make full use of **-bh**, and sometimes you just want to answer the question "Does this address have access?" without bothering with any further details.

The *exim_checkaccess* utility is a "packaged" version of **-bh**. It takes two arguments, an IP address and an email address:

```
exim_checkaccess 10.9.8.7 A.User@a.domain.example
```

The utility runs a call to Exim with the **-bh** option, to test whether the given email address would be accepted in a RCPT command in a TCP/IP connection from the host with the given IP address. The output of the utility is either the word "accepted", or the SMTP error response, for example:

```
Rejected:
550 Relay not permitted
```

When running this test, the utility uses <> as the envelope sender address for the MAIL command, but you can change this by providing additional options. These are passed directly to the Exim command. For example, to specify that the test is to be run with the sender address *himself@there.example* you can use:

```
exim_checkaccess 10.9.8.7 A.User@a.domain.example \
                 -f himself@there.example
```

Note that these additional Exim command line items must be given after the two mandatory arguments.

Because the **exim_checkaccess** uses **-bh**, it does not perform callouts while running its checks. You can run checks that include callouts by using **-bhc**, but this is not yet available in a "packaged" form.

50.9 Making DBM files (exim_dbmbuild)

The *exim_dbmbuild* program reads an input file containing keys and data in the format used by the *lsearch* lookup (see section 9.3). It writes a DBM file using the lower-cased alias names as keys and the remainder of the information as data. The lower-casing 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 zeroes 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 file. It creates the output under a temporary name, and then renames it if all went well.

If the native DB interface is in use (USE_DB is set in a compile-time configuration file – this is common in free versions of Unix) the two file names 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), two files are used, with the 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. This is also the case when the Berkeley functions are used in compatibility mode (though this is not recommended), because in that case it adds a *.db* suffix to the file name.

If a duplicate key is encountered, the program outputs a warning, and when it finishes, its return code is 1 rather than zero, unless the **-noduperr** option is used. By default, only the first of a set of duplicates is used – this makes it compatible with *lsearch* lookups. There is an option **-lastdup** which

causes it to use the data for the last duplicate instead. There is also an option **-nowarn**, which stops it listing duplicate keys to **stderr**. For other errors, where it doesn't actually make a new file, the return code is 2.

50.10 Finding individual retry times (exinext)

A utility called *exinext* (mostly a Perl script) provides the ability to fish specific information out of the retry database. Given a mail domain (or a complete address), it looks up the hosts for that domain, and outputs any retry information for the hosts or for the domain. At present, the retry information is obtained by running *exim_dumpdb* (see below) and post-processing the output. For example:

```
$ exinext piglet@milne.fict.example
kanga.milne.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.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
```

You can also give *exinext* a local part, without a domain, and it will give any retry information for that local part in your default domain. A message id can be used 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 section 45.2). *exinext* is not particularly efficient, but then it is not expected to be run very often.

The *exinext* utility calls Exim to find out information such as the location of the spool directory. The utility has **-C** and **-D** options, which are passed on to the *exim* commands. The first specifies an alternate Exim configuration file, and the second sets macros for use within the configuration file. These features are mainly to help in testing, but might also be useful in environments where more than one configuration file is in use.

50.11 Hints database maintenance

Three utility programs are provided for maintaining the DBM files that Exim uses to contain its delivery hint information. Each program requires two arguments. The first specifies the name of Exim's spool directory, and the second is the name of the database it is to operate on. These are as follows:

- *retry*: the database of retry information
- *wait-*<*transport name*>: databases of information about messages waiting for remote hosts
- *callout*: the callout cache
- *ratelimit*: the data for implementing the ratelimit ACL condition
- *misc*: other hints data

The *misc* database is used for

- Serializing ETRN runs (when **smtp_etrn_serialize** is set)
- Serializing delivery to a specific host (when **serialize_hosts** is set in an *smtp* transport)

50.12 exim_dumpdb

The entire contents of a database are written to the standard output by the *exim_dumpdb* program, which has no options or arguments other than the spool and database names. For example, to dump the retry database:

```
exim_dumpdb /var/spool/exim retry
```

Two lines of output are produced for each entry:

```
T:mail.ref.example:192.168.242.242 146 77 Connection refused
31-Oct-1995 12:00:12 02-Nov-1995 12:21:39 02-Nov-1995 20:21:39 *
```

The first item on the first line is the key of the record. It starts with one of the letters R, or T, depending on whether it refers to a routing or transport retry. For a local delivery, the next part is the local address; for a remote delivery it is the name of the remote host, followed by its failing IP address (unless **retry_include_ip_address** is set false on the *smtp* transport). If the remote port is not the standard one (port 25), it is added to the IP address. Then there follows an error code, an additional error code, and a textual description of the error.

The three times on the second line are the time of first failure, the time of the last delivery attempt, and the computed time for the next attempt. The line ends with an asterisk if the cutoff time for the last retry rule has been exceeded.

Each output line from *exim_dumpdb* for the *wait-xxx* databases consists of a host name followed by a list of ids for messages that are or were waiting to be delivered to that host. If there are a very large number for any one host, continuation records, with a sequence number added to the host name, may be seen. The data in these records is often out of date, because a message may be routed to several alternative hosts, and Exim makes no effort to keep cross-references.

50.13 exim_tidydb

The *exim_tidydb* utility program is used to tidy up the contents of a hints database. If run with no options, it removes all records that are more than 30 days old. The age is calculated from the date and time that the record was last updated. Note that, in the case of the retry database, it is *not* the time since the first delivery failure. Information about a host that has been down for more than 30 days will remain in the database, provided that the record is updated sufficiently often.

The cutoff date can be altered by means of the **-t** option, which must be followed by a time. For example, to remove all records older than a week from the retry database:

```
exim_tidydb -t 7d /var/spool/exim retry
```

Both the *wait-xxx* and *retry* databases contain items that involve message ids. In the former these appear as data in records keyed by host – they were messages that were waiting for that host – and in the latter they are the keys for retry information for messages that have suffered certain types of error. When *exim_tidydb* is run, a check is made to ensure that message ids in database records are those of messages that are still on the queue. Message ids for messages that no longer exist are removed from *wait-xxx* records, and if this leaves any records empty, they are deleted. For the *retry* database, records whose keys are non-existent message ids are removed. The *exim_tidydb* utility outputs comments on the standard output whenever it removes information from the database.

Certain records are automatically removed by Exim when they are no longer needed, but others are not. For example, if all the MX hosts for a domain are down, a retry record is created for each one. If the primary MX host comes back first, its record is removed when Exim successfully delivers to it, but the records for the others remain because Exim has not tried to use those hosts.

It is important, therefore, to run *exim_tidydb* periodically on all the hints databases. You should do this at a quiet time of day, because it requires a database to be locked (and therefore inaccessible to Exim) while it does its work. Removing records from a DBM file does not normally make the file smaller, but all the common DBM libraries are able to re-use the space that is released. After an initial phase of increasing in size, the databases normally reach a point at which they no longer get any bigger, as long as they are regularly tidied.

Warning: If you never run *exim_tidydb*, the space used by the hints databases is likely to keep on increasing.

50.14 exim_fixdb

The *exim_fixdb* program is a utility for interactively modifying databases. Its main use is for testing Exim, but it might also be occasionally useful for getting round problems in a live system. It has no

options, and its interface is somewhat crude. On entry, it prompts for input with a right angle-bracket. A key of a database record can then be entered, and the data for that record is displayed.

If "d" is typed at the next prompt, the entire record is deleted. For all except the *retry* database, that is the only operation that can be carried out. For the *retry* database, each field is output preceded by a number, and data for individual fields can be changed by typing the field number followed by new data, for example:

```
> 4 951102:1000
```

resets the time of the next delivery attempt. Time values are given as a sequence of digit pairs for year, month, day, hour, and minute. Colons can be used as optional separators.

50.15 Mailbox maintenance (exim_lock)

The *exim_lock* utility locks a mailbox file using the same algorithm as Exim. For a discussion of locking issues, see section 26.3. *Exim_lock* can be used to prevent any modification of a mailbox by Exim or a user agent while investigating a problem. 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 (using C's *system()* function); 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. The following options are available:

-fcntl
Use *fcntl()* locking on the open mailbox.

-flock
Use *flock()* locking on the open mailbox, provided the operating system supports it.

-interval
This must be followed by a number, which is a number of seconds; it sets the interval to sleep between retries (default 3).

-lockfile
Create a lock file before opening the mailbox.

-mbx
Lock the mailbox using MBX rules.

-q
Suppress verification output.

-retries
This must be followed by a number; it sets the number of times to try to get the lock (default 10).

-restore_time
This option causes **exim_lock** to restore the modified and read times to the locked file before exiting. This allows you to access a locked mailbox (for example, to take a backup copy) without disturbing the times that the user subsequently sees.

-timeout
This must be followed by a number, which is a number of seconds; it sets a timeout to be used with a blocking *fcntl()* lock. If it is not set (the default), a non-blocking call is used.

-v
Generate verbose output.

If none of **-fcntl**, **-flock**, **-lockfile** or **-mbx** are given, the default is to create a lock file and also to use *fcntl()* locking on the mailbox, which is the same as Exim's default. The use of **-flock** or **-fcntl** requires that the file be writeable; the use of **-lockfile** requires that the directory containing the file be writeable. Locking by lock file does not last for ever; Exim assumes that a lock file is expired if it is more than 30 minutes old.

The **-mbx** option can be used with either or both of **-fcntl** or **-flock**. It assumes **-fcntl** by default. MBX locking causes a shared lock to be taken out on the open mailbox, and an exclusive lock on the

file */tmp/.n.m* where *n* and *m* are the device number and inode number of the mailbox file. When the locking is released, if an exclusive lock can be obtained for the mailbox, the file in */tmp* is deleted.

The default output contains verification of the locking that takes place. The **-v** option causes some additional information to be given. The **-q** option suppresses all output except error messages.

A command such as

```
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
```

runs a specific non-interactive sequence of commands while the file is locked, suppressing all verifi¤ cation output. A single command can be run 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.

51. The Exim monitor

The Exim monitor is an application which displays in an X window information about the state of Exim's queue and 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 to perform any actions requested.

51.1 Running the monitor

The monitor is started by running the script called *eximon*. This is a shell script that sets up a number of environment variables, and then runs the binary called *eximon.bin*. The default appearance of the monitor window can be changed by editing the *Local/eximon.conf* file created by editing *exim_monitor/EDITME*. Comments in that file describe what the various parameters are for.

The parameters that get built into the *eximon* script 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.

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 colour of the background to light grey 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 colour can be changed by setting a resource called "highlight" (an odd name, but that's 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

```
xrdb -merge <<End
Eximon*highlight: gray
End
```

In order to see the contents of messages on the queue, and to operate on them, *eximon* must either be run as root or by an admin user.

The monitor's window is divided into three parts. The first 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 following sections describe these different parts of the display.

51.2 The stripcharts

The first stripchart is always a count of messages on the queue. Its name can be configured by setting QUEUE_STRIPCHART_NAME in the *Local/eximon.conf* file. 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; this can be adjusted by a parameter in the *Local/eximon.conf* file.

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 which shows the percentage fullness of a particular disk partition, which is useful when local deliveries 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%, and the scale is given as "x10%". This chart is configured by setting SIZE_STRIPCHART and (optionally) SIZE_STRIPCHART_NAME in the *Local/eximon.conf* file.

51.3 Main action buttons

Below the stripcharts there 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 default minimum size leaves just the queue count stripchart and these two buttons visible. Pressing the "Size" button causes the window to expand to its maximum size, unless it is already at the maximum, in which case it is reduced to its minimum.

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. When it is expanding from its minimum size, the old position is remembered, and next time it is reduced to the minimum it is moved back there.

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. The idea 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 MIN_HEIGHT and MIN_WIDTH values in *Local/eximon.conf*.

Normally, the monitor starts up with the window at its full size, but it can be built so that it starts up with the window at its smallest size, by setting START_SMALL=yes in *Local/eximon.conf*.

51.4 The log display

The second section of the window is an area in which a display of the tail of the main log is maintained. To save space on the screen, the timestamp on each log line is shortened by removing the date and, if **log_timezone** is set, the timezone. The log tail 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 sub-window has a scroll bar at its lefthand side which can be used to move back to look at earlier text, and the up and down arrow keys also have a scrolling effect. The amount of log that is kept depends on the setting of LOG_BUFFER in *Local/eximon.conf*, which specifies the amount of memory to use. When this is full, the earlier 50% of data is discarded – this is much more efficient than throwing it away line by line. The sub-window also has a horizontal scroll bar 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 parameters in the configuration file *Local/eximon.conf*.

Searches of the text in the log window can be carried out by means of the ^R and ^S keystrokes, which default to a reverse and a forward 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. 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 is moved to the end of the new text.

Pressing ^R or ^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 "Return" key is pressed, a single search is done and the window is closed. If ^C is typed the search is cancelled.

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. The compile-time option EXIMON_TEXTPOP can be unset to cut out the modified **TextPop**, making it possible to build Eximon on these systems, at the expense of having unwanted items in the search popup window.

51.5 The queue display

The bottom section of the monitor window contains a list of all messages that are on the queue, which includes those currently being received or delivered, as well as those awaiting delivery. The size of this subwindow is controlled by parameters in the configuration file *Local/eximon.conf*, and the frequency at which it is updated is controlled by another parameter in the same file – the default is 5 minutes, since queue scans can be quite expensive. However, there is an "Update" action button just above the display which 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 dialogue box called "Hide addresses ending with" is put up. If you type anything in here and press "Return", 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 there is an address that 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, *cam.ac.uk* specifies all addresses in Cambridge, while *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 do not get forgotten, a hide request is automatically cancelled after one hour.

While the dialogue box is displayed, you can't 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 dialogue box, you have to 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 bounce 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 left-hand side.

The queue display has a vertical scroll bar, 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 above 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.

51.6 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, 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 the MENU_EVENT parameter in *Local/eximon.conf* to change the default, or 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
```

The title of the menu is the message id, and it contains entries which 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 that contains the envelope information and headers is displayed in a new text window. See chapter 53 for a description of the format of spool files.

- *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 20,000 bytes to the amount of data displayed. This can be changed by setting the BODY_MAX option at compile time, or the EXIMON_BODY_MAX option at run time.

- *deliver message*: A call to Exim is made using the **-M** option to request delivery of the message. This 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 bounce message 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 a bounce message.

- *add recipient*: A dialog box is displayed into which a recipient address can be typed. If the address is not qualified and the QUALIFY_DOMAIN parameter is set in *Local/eximon.conf*, the address is qualified with that domain. Otherwise it must be entered as a fully qualified address. Pressing RETURN 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. If the address is not qualified and the QUALIFY_DOMAIN parameter is set in *Local/eximon.conf*, the address is qualified with that domain. Otherwise it must be entered as a fully qualified address. Pressing RETURN 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. Pressing RETURN 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. If you want to set an empty sender (as in bounce messages), you must specify it as "<>". Otherwise, if the address is not qualified and the QUALIFY_DOMAIN parameter is set in *Local/eximon.conf*, the address is qualified with that domain.

When a delivery is forced, a window showing the **-v** output is displayed. In other cases when a call to Exim is made, if there is any 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. However, if you set ACTION_OUTPUT=yes in *Local/eximon.conf*, a window showing the Exim command is always opened, even if no output is generated.

The queue display is automatically updated for actions such as freezing and thawing, unless ACTION_QUEUE_UPDATE=no has been set in *Local/eximon.conf*. In this case the "Update" button has to be used to force an update of the display after one of these actions.

In any text window that is displayed as result of a menu action, the normal cut-and-paste facility is available, and searching can be carried out using ^R and ^S, as described above for the log tail window.

52. Security considerations

This chapter discusses a number of issues concerned with security, some of which are also covered in other parts of this manual.

For reasons that this author does not understand, some people have promoted Exim as a "particularly secure" mailer. Perhaps it is because of the existence of this chapter in the documentation. However, the intent of the chapter is simply to describe the way Exim works in relation to certain security concerns, not to make any specific claims about the effectiveness of its security as compared with other MTAs.

What follows is a description of the way Exim is supposed to be. Best efforts have been made to try to ensure that the code agrees with the theory, but an absence of bugs can never be guaranteed. Any that are reported will get fixed as soon as possible.

52.1 Building a more "hardened" Exim

There are a number of build-time options that can be set in *Local/Makefile* to create Exim binaries that are "harder" to attack, in particular by a rogue Exim administrator who does not have the root password, or by someone who has penetrated the Exim (but not the root) account. These options are as follows:

- ALT_CONFIG_PREFIX can be set to a string that is required to match the start of any file names used with the **-C** option. When it is set, these file names are also not allowed to contain the sequence "/../". (However, if the value of the **-C** option is identical to the value of CONFIGURE_FILE in *Local/Makefile*, Exim ignores **-C** and proceeds as usual.) There is no default setting for **ALT_CONFIG_PREFIX**.

  If the permitted configuration files are confined to a directory to which only root has access, this guards against someone who has broken into the Exim account from running a privileged Exim with an arbitrary configuration file, and using it to break into other accounts.

- If ALT_CONFIG_ROOT_ONLY is defined, root privilege is retained for **-C** and **-D** only if the caller of Exim is root. Without it, the Exim user may also use **-C** and **-D** and retain privilege. Setting this option locks out the possibility of testing a configuration using **-C** right through message reception and delivery, even if the caller is root. The reception works, but by that time, Exim is running as the Exim user, so when it re-execs to regain privilege for the delivery, the use of **-C** causes privilege to be lost. However, root can test reception and delivery using two separate commands. ALT_CONFIG_ROOT_ONLY is not set by default.

- If DISABLE_D_OPTION is defined, the use of the **-D** command line option is disabled.

- FIXED_NEVER_USERS can be set to a colon-separated list of users that are never to be used for any deliveries. This is like the **never_users** runtime option, but it cannot be overridden; the runtime option adds additional users to the list. The default setting is "root"; this prevents a non-root user who is permitted to modify the runtime file from using Exim as a way to get root.

52.2 Root privilege

The Exim binary is normally setuid to root, which means that it gains root privilege (runs as root) when it starts execution. In some special cases (for example, when the daemon is not in use and there are no local deliveries), it may be possible to run Exim setuid to some user other than root. This is discussed in the next section. However, in most installations, root privilege is required for two things:

- To set up a socket connected to the standard SMTP port (25) when initialising the listening daemon. If Exim is run from *inetd*, this privileged action is not required.

- To be able to change uid and gid in order to read users' *.forward* files and perform local deliveries as the receiving user or as specified in the configuration.

It is not necessary to be root to do any of the other things Exim does, such as receiving messages and delivering them externally over SMTP, and it is obviously more secure if Exim does not run as root

except when necessary. For this reason, a user and group for Exim to use must be defined in *Local/Makefile*. These are known as "the Exim user" and "the Exim group". Their values can be changed by the run time configuration, though this is not recommended. Often a user called *exim* is used, but some sites use *mail* or another user name altogether.

Exim uses *setuid()* whenever it gives up root privilege. This is a permanent abdication; the process cannot regain root afterwards. Prior to release 4.00, *seteuid()* was used in some circumstances, but this is no longer the case.

After a new Exim process has interpreted its command line options, it changes uid and gid in the following cases:

- If the **-C** option is used to specify an alternate configuration file, or if the **-D** option is used to define macro values for the configuration, and the calling process is not running as root or the Exim user, the uid and gid are changed to those of the calling process. However, if ALT_CONFIG_ROOT_ONLY is defined in *Local/Makefile*, only root callers may use **-C** and **-D** without losing privilege, and if DISABLE_D_OPTION is set, the **-D** option may not be used at all.
- If the expansion test option (**-be**) or one of the filter testing options (**-bf** or **-bF**) are used, the uid and gid are changed to those of the calling process.
- If the process is not a daemon process or a queue runner process or a delivery process or a process for testing address routing (started with **-bt**), the uid and gid are changed to the Exim user and group. This means that Exim always runs under its own uid and gid when receiving messages. This also applies when testing address verification (the **-bv** option) and testing incoming message policy controls (the **-bh** option).
- For a daemon, queue runner, delivery, or address testing process, the uid remains as root at this stage, but the gid is changed to the Exim group.

The processes that initially retain root privilege behave as follows:

- A daemon process changes the gid to the Exim group and the uid to the Exim user after setting up one or more listening sockets. The *initgroups()* function is called, so that if the Exim user is in any additional groups, they will be used during message reception.
- A queue runner process retains root privilege throughout its execution. Its job is to fork a controlled sequence of delivery processes.
- A delivery process retains root privilege throughout most of its execution, but any actual deliveries (that is, the transports themselves) are run in subprocesses which always change to a non-root uid and gid. For local deliveries this is typically the uid and gid of the owner of the mailbox; for remote deliveries, the Exim uid and gid are used. Once all the delivery subprocesses have been run, a delivery process changes to the Exim uid and gid while doing post-delivery tidying up such as updating the retry database and generating bounce and warning messages.

  While the recipient addresses in a message are being routed, the delivery process runs as root. However, if a user's filter file has to be processed, this is done in a subprocess that runs under the individual user's uid and gid. A system filter is run as root unless **system_filter_user** is set.
- A process that is testing addresses (the **-bt** option) runs as root so that the routing is done in the same environment as a message delivery.

52.3 Running Exim without privilege

Some installations like to run Exim in an unprivileged state for more of its operation, for added security. Support for this mode of operation is provided by the global option **deliver_drop_privilege**. When this is set, the uid and gid are changed to the Exim user and group at the start of a delivery process (and also queue runner and address testing processes). This means that address routing is no longer run as root, and the deliveries themselves cannot change to any other uid.

Leaving the binary setuid to root, but setting **deliver_drop_privilege** means that the daemon can still be started in the usual way, and it can respond correctly to SIGHUP because the re-invocation regains root privilege.

An alternative approach is to make Exim setuid to the Exim user and also setgid to the Exim group. If you do this, the daemon must be started from a root process. (Calling Exim from a root process makes it behave in the way it does when it is setuid root.) However, the daemon cannot restart itself after a SIGHUP signal because it cannot regain privilege.

It is still useful to set **deliver_drop_privilege** in this case, because it stops Exim from trying to re-invoke itself to do a delivery after a message has been received. Such a re-invocation is a waste of resources because it has no effect.

If restarting the daemon is not an issue (for example, if **mua_wrapper** is set, or *inetd* is being used instead of a daemon), having the binary setuid to the Exim user seems a clean approach, but there is one complication:

In this style of operation, Exim is running with the real uid and gid set to those of the calling process, and the effective uid/gid set to Exim's values. Ideally, any association with the calling process' uid/gid should be dropped, that is, the real uid/gid should be reset to the effective values so as to discard any privileges that the caller may have. While some operating systems have a function that permits this action for a non-root effective uid, quite a number of them do not. Because of this lack of standardization, Exim does not address this problem at this time.

For this reason, the recommended approach for "mostly unprivileged" running is to keep the Exim binary setuid to root, and to set **deliver_drop_privilege**. This also has the advantage of allowing a daemon to be used in the most straightforward way.

If you configure Exim not to run delivery processes as root, there are a number of restrictions on what you can do:

- You can deliver only as the Exim user/group. You should explicitly use the **user** and **group** options to override routers or local transports that normally deliver as the recipient. This makes sure that configurations that work in this mode function the same way in normal mode. Any implicit or explicit specification of another user causes an error.
- Use of *.forward* files is severely restricted, such that it is usually not worthwhile to include them in the configuration.
- Users who wish to use *.forward* would have to make their home directory and the file itself accessible to the Exim user. Pipe and append-to-file entries, and their equivalents in Exim filters, cannot be used. While they could be enabled in the Exim user's name, that would be insecure and not very useful.
- Unless the local user mailboxes are all owned by the Exim user (possible in some POP3 or IMAP-only environments):

  (1) They must be owned by the Exim group and be writeable by that group. This implies you must set **mode** in the appendfile configuration, as well as the mode of the mailbox files themselves.

  (2) You must set **no_check_owner**, since most or all of the files will not be owned by the Exim user.

  (3) You must set **file_must_exist**, because Exim cannot set the owner correctly on a newly created mailbox when unprivileged. This also implies that new mailboxes need to be created manually.

These restrictions severely restrict what can be done in local deliveries. However, there are no restrictions on remote deliveries. If you are running a gateway host that does no local deliveries, setting **deliver_drop_privilege** gives more security at essentially no cost.

If you are using the **mua_wrapper** facility (see chapter 48), **deliver_drop_privilege** is forced to be true.

52.4 Delivering to local files

Full details of the checks applied by *appendfile* before it writes to a file are given in chapter 26.

52.5 IPv4 source routing

Many operating systems suppress IP source-routed packets in the kernel, but some cannot be made to do this, so Exim does its own check. It logs incoming IPv4 source-routed TCP calls, and then drops them. Things are all different in IPv6. No special checking is currently done.

52.6 The VRFY, EXPN, and ETRN commands in SMTP

Support for these SMTP commands is disabled by default. If required, they can be enabled by defining suitable ACLs.

52.7 Privileged users

Exim recognizes two sets of users with special privileges. Trusted users are able to submit new messages to Exim locally, but supply their own sender addresses and information about a sending host. For other users submitting local messages, Exim sets up the sender address from the uid, and doesn't permit a remote host to be specified.

However, an untrusted user is permitted to use the **-f** command line option in the special form **-f <>** to indicate that a delivery failure for the message should not cause an error report. This affects the message's envelope, but it does not affect the *Sender:* header. Untrusted users may also be permitted to use specific forms of address with the **-f** option by setting the **untrusted_set_sender** option.

Trusted users are used to run processes that receive mail messages from some other mail domain and pass them on to Exim for delivery either locally, or over the Internet. Exim trusts a caller that is running as root, as the Exim user, as any user listed in the **trusted_users** configuration option, or under any group listed in the **trusted_groups** option.

Admin users are permitted to do things to the messages on Exim's queue. They can freeze or thaw messages, cause them to be returned to their senders, remove them entirely, or modify them in various ways. In addition, admin users can run the Exim monitor and see all the information it is capable of providing, which includes the contents of files on the spool.

By default, the use of the **-M** and **-q** options to cause Exim to attempt delivery of messages on its queue is restricted to admin users. This restriction can be relaxed by setting the **no_prod_requires_admin** option. Similarly, the use of **-bp** (and its variants) to list the contents of the queue is also restricted to admin users. This restriction can be relaxed by setting **no_queue_list_requires_admin**.

Exim recognizes an admin user if the calling process is running as root or as the Exim user or if any of the groups associated with the calling process is the Exim group. It is not necessary actually to be running under the Exim group. However, if admin users who are not root or the Exim user are to access the contents of files on the spool via the Exim monitor (which runs unprivileged), Exim must be built to allow group read access to its spool files.

52.8 Spool files

Exim's spool directory and everything it contains is owned by the Exim user and set to the Exim group. The mode for spool files is defined in the *Local/Makefile* configuration file, and defaults to 0640. This means that any user who is a member of the Exim group can access these files.

52.9 Use of argv[0]

Exim examines the last component of **argv[0]**, and if it matches one of a set of specific strings, Exim assumes certain options. For example, calling Exim with the last component of **argv[0]** set to "rsmtp" is exactly equivalent to calling it with the option **-bS**. There are no security implications in this.

52.10 Use of %f formatting

The only use made of "%f" by Exim is in formatting load average values. These are actually stored in integer variables as 1000 times the load average. Consequently, their range is limited and so therefore is the length of the converted output.

52.11 Embedded Exim path

Exim uses its own path name, which is embedded in the code, only when it needs to re-exec in order to regain root privilege. Therefore, it is not root when it does so. If some bug allowed the path to get overwritten, it would lead to an arbitrary program's being run as exim, not as root.

52.12 Use of sprintf()

A large number of occurrences of "sprintf" in the code are actually calls to *string_sprintf()*, a function that returns the result in malloc'd store. The intermediate formatting is done into a large fixed buffer by a function that runs through the format string itself, and checks the length of each conversion before performing it, thus preventing buffer overruns.

The remaining uses of *sprintf()* happen in controlled circumstances where the output buffer is known to be sufficiently long to contain the converted string.

52.13 Use of debug_printf() and log_write()

Arbitrary strings are passed to both these functions, but they do their formatting by calling the function *string_vformat()*, which runs through the format string itself, and checks the length of each conversion.

52.14 Use of strcat() and strcpy()

These are used only in cases where the output buffer is known to be large enough to hold the result.

53. Format of spool files

A message on Exim's queue consists of two files, whose names are the message id followed by -D and -H, respectively. The data portion of the message is kept in the -D file on its own. The message's envelope, status, and headers are all kept in the -H file, whose format is described in this chapter. Each of these two files contains the final component of its own name as its first line. This is insurance against disk crashes where the directory is lost but the files themselves are recoverable.

Some people are tempted into editing -D files in order to modify messages. You need to be extremely careful if you do this; it is not recommended and you are on your own if you do it. Here are some of the pitfalls:

- You must ensure that Exim does not try to deliver the message while you are fiddling with it. The safest way is to take out a write lock on the -D file, which is what Exim itself does, using *fcntl()*. If you update the file in place, the lock will be retained. If you write a new file and rename it, the lock will be lost at the instant of rename.
- If you change the number of lines in the file, the value of *$body_linecount*, which is stored in the -H file, will be incorrect. At present, this value is not used by Exim, but there is no guarantee that this will always be the case.
- If the message is in MIME format, you must take care not to break it.
- If the message is cryptographically signed, any change will invalidate the signature.

All in all, modifying -D files is fraught with danger.

Files whose names end with -J may also be seen in the *input* directory (or its subdirectories when **split_spool_directory** is set). These are journal files, used to record addresses to which the message has been delivered during the course of a delivery attempt. If there are still undelivered recipients at the end, the -H file is updated, and the -J file is deleted. If, however, there is some kind of crash (for example, a power outage) before this happens, the -J file remains in existence. When Exim next processes the message, it notices the -J file and uses it to update the -H file before starting the next delivery attempt.

53.1 Format of the -H file

The second line of the -H file contains the login name for the uid of the process that called Exim to read the message, followed by the numerical uid and gid. For a locally generated message, this is normally the user who sent the message. For a message received over TCP/IP via the daemon, it is normally the Exim user.

The third line of the file contains the address of the message's sender as transmitted in the envelope, contained in angle brackets. The sender address is empty for bounce messages. For incoming SMTP mail, the sender address is given in the MAIL command. For locally generated mail, the sender address is created by Exim from the login name of the current user and the configured **qualify_domain**. However, this can be overridden by the **-f** option or a leading "From " line if the caller is trusted, or if the supplied address is "<>" or an address that matches **untrusted_set_senders**.

The fourth line contains two numbers. The first is the time that the message was received, in the conventional Unix form – the number of seconds since the start of the epoch. The second number is a count of the number of messages warning of delayed delivery that have been sent to the sender.

There follow a number of lines starting with a hyphen. These can appear in any order, and are omitted when not relevant:

-acl <*number*> <*length*>
: This item is obsolete, and is not generated from Exim release 4.61 onwards; **-aclc** and **-aclm** are used instead. However, **-acl** is still recognized, to provide backward compatibility. In the old format, a line of this form is present for every ACL variable that is not empty. The number identifies the variable; the **acl_cx** variables are numbered 0–9 and the **acl_mx** variables are numbered 10–19. The length is the length of the data string for the variable. The string itself starts at

the beginning of the next line, and is followed by a newline character. It may contain internal newlines.

-aclc <*rest-of-name*> <*length*>
A line of this form is present for every ACL connection variable that is defined. Note that there is a space between **-aclc** and the rest of the name. The length is the length of the data string for the variable. The string itself starts at the beginning of the next line, and is followed by a newline character. It may contain internal newlines.

-aclm <*rest-of-name*> <*length*>
A line of this form is present for every ACL message variable that is defined. Note that there is a space between **-aclm** and the rest of the name. The length is the length of the data string for the variable. The string itself starts at the beginning of the next line, and is followed by a newline character. It may contain internal newlines.

-active_hostname <*hostname*>
This is present if, when the message was received over SMTP, the value of *$smtp_active_hostname* was different to the value of *$primary_hostname*.

-allow_unqualified_recipient
This is present if unqualified recipient addresses are permitted in header lines (to stop such addresses from being qualified if rewriting occurs at transport time). Local messages that were input using **-bnq** and remote messages from hosts that match **recipient_unqualified_hosts** set this flag.

-allow_unqualified_sender
This is present if unqualified sender addresses are permitted in header lines (to stop such addresses from being qualified if rewriting occurs at transport time). Local messages that were input using **-bnq** and remote messages from hosts that match **sender_unqualified_hosts** set this flag.

-auth_id <*text*>
The id information for a message received on an authenticated SMTP connection – the value of the *$authenticated_id* variable.

-auth_sender <*address*>
The address of an authenticated sender – the value of the *$authenticated_sender* variable.

-body_linecount <*number*>
This records the number of lines in the body of the message, and is always present.

-body_zerocount <*number*>
This records the number of binary zero bytes in the body of the message, and is present if the number is greater than zero.

-deliver_firsttime
This is written when a new message is first added to the spool. When the spool file is updated after a deferral, it is omitted.

-frozen <*time*>
The message is frozen, and the freezing happened at <*time*>.

-helo_name <*text*>
This records the host name as specified by a remote host in a HELO or EHLO command.

-host_address <*address*>.<*port*>
This records the IP address of the host from which the message was received and the remote port number that was used. It is omitted for locally generated messages.

-host_auth <*text*>
If the message was received on an authenticated SMTP connection, this records the name of the authenticator – the value of the *$sender_host_authenticated* variable.

-host_lookup_failed
This is present if an attempt to look up the sending host's name from its IP address failed. It corresponds to the *$host_lookup_failed* variable.

-host_name <*text*>
This records the name of the remote host from which the message was received, if the host name was looked up from the IP address when the message was being received. It is not present if no reverse lookup was done.

-ident <*text*>
For locally submitted messages, this records the login of the originating user, unless it was a trusted user and the **-oMt** option was used to specify an ident value. For messages received over TCP/IP, this records the ident string supplied by the remote host, if any.

-interface_address <*address*>.<*port*>
This records the IP address of the local interface and the port number through which a message was received from a remote host. It is omitted for locally generated messages.

-local
The message is from a local sender.

-localerror
The message is a locally-generated bounce message.

-local_scan <*string*>
This records the data string that was returned by the *local_scan()* function when the message was received – the value of the *$local_scan_data* variable. It is omitted if no data was returned.

-manual_thaw
The message was frozen but has been thawed manually, that is, by an explicit Exim command rather than via the auto-thaw process.

-N
A testing delivery process was started using the **-N** option to suppress any actual deliveries, but delivery was deferred. At any further delivery attempts, **-N** is assumed.

-received_protocol
This records the value of the *$received_protocol* variable, which contains the name of the protocol by which the message was received.

-sender_set_untrusted
The envelope sender of this message was set by an untrusted local caller (used to ensure that the caller is displayed in queue listings).

-spam_score_int <*number*>
If a message was scanned by SpamAssassin, this is present. It records the value of *$spam_score_int*.

-tls_certificate_verified
A TLS certificate was received from the client that sent this message, and the certificate was verified by the server.

-tls_cipher <*cipher name*>
When the message was received over an encrypted connection, this records the name of the cipher suite that was used.

-tls_peerdn <*peer DN*>
When the message was received over an encrypted connection, and a certificate was received from the client, this records the Distinguished Name from that certificate.

Following the options there is a list of those addresses to which the message is not to be delivered. This set of addresses is initialized from the command line when the **-t** option is used and **extract_addresses_remove_arguments** is set; otherwise it starts out empty. Whenever a successful delivery is made, the address is added to this set. The addresses are kept internally as a balanced binary tree, and it is a representation of that tree which is written to the spool file. If an address is expanded via an alias or forward file, the original address is added to the tree when deliveries to all its child addresses are complete.

If the tree is empty, there is a single line in the spool file containing just the text "XX". Otherwise, each line consists of two letters, which are either Y or N, followed by an address. The address is the value for the node of the tree, and the letters indicate whether the node has a left branch and/or a right branch attached to it, respectively. If branches exist, they immediately follow. Here is an example of a three-node tree:

```
YY darcy@austen.fict.example
NN alice@wonderland.fict.example
NN editor@thesaurus.ref.example
```

After the non-recipients tree, there is a list of the message's recipients. This is a simple list, preceded by a count. It includes all the original recipients of the message, including those to whom the message has already been delivered. In the simplest case, the list contains one address per line. For example:

```
4
editor@thesaurus.ref.example
darcy@austen.fict.example
rdo@foundation
alice@wonderland.fict.example
```

However, when a child address has been added to the top-level addresses as a result of the use of the **one_time** option on a *redirect* router, each line is of the following form:

<*top-level address*> <*errors_to address*> <*length*>,<*parent number*>#<*flag bits*>

The 01 flag bit indicates the presence of the three other fields that follow the top-level address. Other bits may be used in future to support additional fields. The <*parent number*> is the offset in the recipients list of the original parent of the "one time" address. The first two fields are the envelope sender that is associated with this address and its length. If the length is zero, there is no special envelope sender (there are then two space characters in the line). A non-empty field can arise from a *redirect* router that has an **errors_to** setting.

A blank line separates the envelope and status information from the headers which follow. A header may occupy several lines of the file, and to save effort when reading it in, each header is preceded by a number and an identifying character. The number is the number of characters in the header, including any embedded newlines and the terminating newline. The character is one of the following:

<*blank*>	header in which Exim has no special interest
B	*Bcc:* header
C	*Cc:* header
F	*From:* header
I	*Message-id:* header
P	*Received:* header – P for "postmark"
R	*Reply-To:* header
S	*Sender:* header
T	*To:* header
*	replaced or deleted header

Deleted or replaced (rewritten) headers remain in the spool file for debugging purposes. They are not transmitted when the message is delivered. Here is a typical set of headers:

```
111P Received: by hobbit.fict.example with local (Exim 4.00)
id 14y9EI-00026G-00; Fri, 11 May 2001 10:28:59 +0100
049  Message-Id: <E14y9EI-00026G-00@hobbit.fict.example>
038* X-rewrote-sender: bb@hobbit.fict.example
042* From: Bilbo Baggins <bb@hobbit.fict.example>
049F From: Bilbo Baggins <B.Baggins@hobbit.fict.example>
099* To: alice@wonderland.fict.example, rdo@foundation,
darcy@austen.fict.example, editor@thesaurus.ref.example
104T To: alice@wonderland.fict.example, rdo@foundation.example,
darcy@austen.fict.example, editor@thesaurus.ref.example
038  Date: Fri, 11 May 2001 10:28:59 +0100
```

The asterisked headers indicate that the envelope sender, *From:* header, and *To:* header have been rewritten, the last one because routing expanded the unqualified domain *foundation*.

54. Adding new drivers or lookup types

The following actions have to be taken in order to add a new router, transport, authenticator, or lookup type to Exim:

(1) Choose a name for the driver or lookup type that does not conflict with any existing name; I will use "newdriver" in what follows.

(2) Add to *src/EDITME* the line:

```
<type>_NEWDRIVER=yes
```

where <*type*> is ROUTER, TRANSPORT, AUTH, or LOOKUP. If the code is not to be included in the binary by default, comment this line out. You should also add any relevant comments about the driver or lookup type.

(3) Add to *src/config.h.defaults* the line:

```
#define <type>_NEWDRIVER
```

(4) Edit *src/drtables.c*, adding conditional code to pull in the private header and create a table entry as is done for all the other drivers and lookup types.

(5) Edit *Makefile* in the appropriate sub-directory (*src/routers*, *src/transports*, *src/auths*, or *src/lookups*); add a line for the new driver or lookup type and add it to the definition of OBJ.

(6) Create *newdriver.h* and *newdriver.c* in the appropriate sub-directory of *src*.

(7) Edit *scripts/MakeLinks* and add commands to link the *.h* and *.c* files as for other drivers and lookups.

Then all you need to do is write the code! A good way to start is to make a proforma by copying an existing module of the same type, globally changing all occurrences of the name, and cutting out most of the code. Note that any options you create must be listed in alphabetical order, because the tables are searched using a binary chop procedure.

There is a *README* file in each of the sub-directories of *src* describing the interface that is expected.

Options index

Symbols

-- 28
--help 28
-B 28
-bd 28
-bdf 29
-be 29, 98, 427
-bem 29, 98
-bF 29, 427
-bf 29, 427
-bfd 30
-bfl 30
-bfp 30
-bfs 30
-bh 30, 323, 427
-bhc 30
-bi 31, 150
-bm 31
-bnq 31
-bP 31
-bp 32, 169
-bpa 32
-bpc 32
-bpr 33
-bpra 33
-bpru 33
-bpu 33
-brt 33
-brw 33
-bS 33
-bs 34
-bt 34, 186
-bV 34
-bv 35, 198, 427
-bvs 35
-C 35, 427
-D 36, 427
-d 36
-dd 37
-dropcr 37
-E 37
-e*x* 37
-F 38
-f 38, 429
  for address testing 34
  for filter testing 30
  overriding “From” line 31
-G 38
-h 38
-i 38
-M 38, 168, 429
-m 40
-Mar 38
-MC 39
-Mc 39
-MCA 39
-MCP 39
-MCQ 39
-MCS 39
-MCT 39
-Mes 39
-Mf 39
-Mg 39
-Mmad 40
-Mmd 40
-Mrm 40
-Mset 40
-Mt 40
-Mvb 40
-Mvc 40
-Mvh 40
-Mvl 40
-N 40
-n 40
-O 41
-oA 41
-oB 41
-odb 41
-odf 41
-odi 41
-odq 41
-odqs 41
-oee 42
-oem 42
-oep 42
-oeq 42
-oew 42
-oi 42
-oitrue 42
-om 43
-oMa 42
-oMaa 42
-oMai 42
-oMas 43
-oMi 43
-oMr 43
-oMs 43
-oMt 43
-oo 43
-oP 43
-or 43
-os 43, 178
-ov 43
-oX 43
-p 44
-pd 44
-ps 44
-q 44, 168, 429
-qf 45
-qff 45
-qi 45
-ql 45
-qq 44
-qR 45
-qS 45
-R 45, 168
-r 46
-S 46
-t 46, 157
-ti 47
-tls-on-connect 47
-Tqt 46
-U 47
-v 47

-x 47

A

accept_8bitmime 146
acl_not_smtp 147
acl_not_smtp_mime 147, 356
acl_not_smtp_start 147
acl_smtp_auth 147
acl_smtp_connect 147
acl_smtp_data 147
acl_smtp_etrn 147
acl_smtp_expn 147
acl_smtp_helo 147
acl_smtp_mail 147
acl_smtp_mailauth 148
acl_smtp_mime 148, 356
acl_smtp_predata 148
acl_smtp_quit 148
acl_smtp_rcpt 148
acl_smtp_starttls 148
acl_smtp_vrfy 148
address_data 186
address_retry_include_sender 266
address_test 186
admin_groups 148
allow_commands 260
allow_defer 220
allow_domain_literals 148
allow_fail 220
allow_fifo 239
allow_filter 220
allow_freeze 221
allow_localhost 267
allow_mx_to_ip 149
allow_symlink 239
allow_utf8_domains 149
auth_advertise_hosts 149
authenticated_sender 267
authenticated_sender_force 267
auto_thaw 149
av_scanner 150, 351

B

batch_id 236, 239, 256, 260
batch_max 236, 239, 256, 260
bcc 253
bi_command 150
body_only 230
bounce_message_file 150
bounce_message_text 150
bounce_return_body 150
bounce_return_message 150
bounce_return_size_limit 150
bounce_sender_authentication 151

C

callout_domain_negative_expire 151
callout_domain_positive_expire 151
callout_negative_expire 151
callout_positive_expire 151
callout_random_local_part 151
cannot_route_message 186
caseful_local_part 187
cc 254
check_ancestor 221
check_group 221, 239
check_local_user 187
check_log_inodes 151
check_log_space 151
check_owner 221, 240
check_rfc2047_length 152
check_secondary_mx 200
check_spool_inodes 152
check_spool_space 152
check_srv 200
check_string 240, 260
client_condition 290
client_domain 304
client_ignore_invalid_base64 297
client_name 299
client_password 304
client_secret 300
client_send 297
client_username 304
command 214, 256, 261
command_group 214
command_timeout 267
command_user 214
condition 187
connect_timeout 267
connection_max_messages 267
create_directory 240
create_file 240
current_directory 214, 230

D

daemon_smtp_ports 152
daemon_startup_retries 152
daemon_startup_sleep 153
data 221
data_timeout 268
debug_print 188, 230
delay_after_cutoff 268, 287
delay_warning 153
delay_warning_condition 121, 153
deliver_drop_privilege 153
deliver_queue_load_max 153
delivery_date_add 230
delivery_date_remove 153, 378
directory 240
directory_file 241
directory_mode 241
directory_transport 222
disable_fsync 154
disable_ipv6 139, 154
disable_logging 188, 230
dns_again_means_nonexist 154
dns_check_names_pattern 154
dns_csa_search_limit 155
dns_csa_use_reverse 155
dns_ipv4_lookup 155
dns_qualify_single 268
dns_retrans 155
dns_retry 155
dns_search_parents 268
domains 188
driver 188, 230, 290
drop_cr 155

dsn_from 155

E

envelope_to_add 230
envelope_to_remove 155, 378
environment 261
errors_copy 156
errors_reply_to 156, 247
errors_to 188, 392, 394
escape_string 241, 261
exim_group 156
exim_path 156
exim_user 157
expn 189
extra_local_interfaces 157
extract_addresses_remove_arguments 157

F

fail_verify 189
fail_verify_recipient 189
fail_verify_sender 189
fallback_hosts 189, 268
file 222, 241, 254
file_expand 254
file_format 241
file_must_exist 242
file_optional 254
file_transport 222
filter_prepend_home 222
final_timeout 269
finduser_retries 157
forbid_blackhole 222
forbid_exim_filter 222
forbid_file 222
forbid_filter_dlfunc 223
forbid_filter_existstest 223
forbid_filter_logwrite 223
forbid_filter_lookup 223
forbid_filter_perl 223
forbid_filter_readfile 223
forbid_filter_readsocket 223
forbid_filter_reply 223
forbid_filter_run 223
forbid_include 223
forbid_pipe 224
forbid_sieve_filter 224
forbid_smtp_code 224
freeze_exec_fail 261
freeze_tell 157
from 254

G

gecos_name 158
gecos_pattern 158
gethostbyname 269
gnutls_require_kx 158, 269, 308
gnutls_require_mac 158, 269, 308
gnutls_require_protocols 158, 269, 308
group 190, 231

H

header_line_maxsize 159
header_maxsize 158
headers 254
headers_add 190, 231
headers_charset 158
headers_only 231
headers_remove 190, 231
headers_rewrite 231
helo_accept_junk_hosts 159
helo_allow_chars 159
helo_data 269
helo_lookup_domains 159
helo_try_verify_hosts 159
helo_verify_hosts 160
hide_child_in_errmsg 224
hold_domains 160
home_directory 231
host_all_ignored 207
host_find_failed 207
host_lookup 160
host_lookup_order 160
host_reject_connection 161
hosts 205, 269
hosts_avoid_esmtp 270
hosts_avoid_pipelining 270
hosts_avoid_tls 270
hosts_connection_nolog 161
hosts_max_try 270
hosts_max_try_hardlimit 270
hosts_nopass_tls 270
hosts_override 271
hosts_randomize 208, 271
hosts_require_auth 271
hosts_require_tls 271
hosts_treat_as_local 88, 161
hosts_try_auth 271

I

ibase_servers 161
ignore_bounce_errors_after 10, 161
ignore_eacces 224
ignore_enotdir 224
ignore_fromline_hosts 162
ignore_fromline_local 162
ignore_quota 256
ignore_status 261
ignore_target_hosts 190
include_directory 224
initgroups 191, 228, 232
interface 271

K

keep_malformed 162
keepalive 272

L

ldap_default_servers 162
ldap_version 162
lmtp_ignore_quota 272
local_from_check 162
local_from_prefix 163
local_from_suffix 163
local_interfaces 163
local_part_prefix 191
local_part_prefix_optional 192

local_part_suffix 192
local_part_suffix_optional 192
local_parts 192
local_scan_timeout 163
local_sender_retain 163
localhost_number 9, 164
lock_fcntl_timeout 242
lock_flock_timeout 242
lock_interval 242
lock_retries 242
lockfile_mode 242
lockfile_timeout 242
log 254
log_as_local 192
log_defer_output 261
log_fail_output 261
log_file_path 164
log_output 262
log_selector 164
log_timezone 164
lookup_open_max 164

M

mailbox_filecount 243
mailbox_size 243
maildir_format 243
maildir_quota_directory_regex 243
maildir_retries 243
maildir_tag 243
maildir_use_size_file 243
maildirfolder_create_regex 244
mailstore_format 244
mailstore_prefix 244
mailstore_suffix 244
max_output 262
max_rcpt 272
max_username_length 165
mbx_format 244
message_body_newlines 125, 165
message_body_visible 125, 165
message_id_header_domain 165
message_id_header_text 165, 379
message_logs 165
message_prefix 244, 262
message_size_limit 165, 232
message_suffix 245, 262
mode 245, 254
mode_fail_narrower 245
modemask 224
more 193, 195, 207
move_frozen_messages 166
mua_wrapper 166, 399
multi_domain 272
mx_domains 201
mx_fail_domains 201
mysql_servers 166

N

never_mail 254
never_users 166
no_*xxx* 52
not_*xxx* 52
notify_comsat 245

O

once 254
once_file_size 255
once_repeat 255
one_time 225
optional 205
oracle_servers 166
owners 225
owngroups 225

P

pass_on_timeout 193
pass_router 193
path 262
percent_hack_domains 167
perl_at_start 135, 167
perl_startup 135, 167
pgsql_servers 167
pid_file_path 167
pipe_as_creator 262
pipe_transport 225
pipelining_advertise_hosts 167
port 205, 272
preserve_message_logs 168, 411
primary_hostname 88, 168
print_topbitchars 168
process_log_path 168
prod_requires_admin 168
protocol 205, 272
public_name 290

Q

qualify_domain 168, 225, 376
qualify_preserve_domain 226
qualify_recipient 169, 376
qualify_single 201
query 205
queue_domains 169
queue_list_requires_admin 169
queue_only 169, 328
queue_only_file 169
queue_only_load 169
queue_only_load_latch 170
queue_only_override 170
queue_run_in_order 170
queue_run_max 170
queue_smtp_domains 170
quota 245
quota_directory 246
quota_filecount 246
quota_is_inclusive 246
quota_size_regex 246
quota_warn_message 156, 246
quota_warn_threshold 247

R

rcpt_include_affixes 232
receive_timeout 171
received_header_text 171
received_headers_max 171
recipient_unqualified_hosts 171
recipients_max 172
recipients_max_reject 172

redirect_router 193
remote_max_parallel 172
remote_sort_domains 172
repeat_use 226
reply_to 255
reply_transport 226
require_files 193
reroute 205
response_pattern 206
restrict_to_path 263
retry_data_expire 173, 287
retry_include_ip_address 273
retry_interval_max 173, 286
retry_use_local_part 194, 232
return_fail_output 263
return_message 255
return_output 263
return_path 232, 394
return_path_add 233
return_path_remove 173, 379
return_size_limit 173
rewrite 226
rewrite_headers 201
rfc1413_hosts 173
rfc1413_query_timeout 173
route_data 208
route_list 208
router_home_directory 195

S

same_domain_copy_routing 202, 208
search_parents 202
self 193, 195
  in *dnslookup* router 200
  in *ipliteral* router 204
  in *manualroute* router 210
  value of host name 129
sender_unqualified_hosts 173
senders 196
serialize_hosts 273
server_advertise_condition 290
server_condition 291, 295
server_debug_print 291
server_hostname 301
server_mail_auth_condition 291
server_mech 301
server_password 304
server_prompts 295
server_realm 301
server_secret 299
server_service 302
server_set_id 291
server_socket 303
shadow_condition 233
shadow_transport 233
sieve_subaddress 226
sieve_useraddress 226
sieve_vacation_directory 226
size_addition 273
skip_syntax_errors 226
smtp_accept_keepalive 173
smtp_accept_max 174
smtp_accept_max_nonmail 174
smtp_accept_max_nonmail_hosts 174
smtp_accept_max_per_connection 174
smtp_accept_max_per_host 174
smtp_accept_queue 175
smtp_accept_queue_per_connection 175
smtp_accept_reserve 175
smtp_active_hostname 175
smtp_banner 176, 315, 325
smtp_check_spool_space 176
smtp_connect_backlog 176
smtp_enforce_sync 176
smtp_etrn_command 122, 176, 388
smtp_etrn_serialize 177
smtp_load_reserve 177
smtp_max_synprot_errors 177
smtp_max_unknown_commands 177
smtp_ratelimit_* 339
smtp_ratelimit_hosts 177
smtp_ratelimit_mail 178
smtp_ratelimit_rcpt 178
smtp_receive_timeout 178
smtp_reserve_hosts 178
smtp_return_error_details 178
socket 256
spamd_address 179, 354
split_spool_directory 179
spool_directory 179
sqlite_lock_timeout 179
srv_fail_domains 202
strict_acl_vars 180, 321
strip_excess_angle_brackets 180
strip_trailing_dot 180
subject 255
syntax_errors_text 227
syntax_errors_to 227
syslog_duplication 180
syslog_facility 180
syslog_processname 180
syslog_timestamp 180
system_filter 180
system_filter_directory_transport 181
system_filter_file_transport 181
system_filter_group 181
system_filter_pipe_transport 181
system_filter_reply_transport 181
system_filter_user 181

T

tcp_nodelay 181
temp_errors 263
text 255
timeout 206, 214, 256, 263
timeout_defer 263
timeout_frozen_after 10, 181
timezone 182
tls_advertise_hosts 182
tls_certificate 182, 273
tls_crl 182, 273
tls_dhparam 182
tls_on_connect_ports 182
tls_privatekey 183, 274
tls_remember_esmtp 183
tls_require_ciphers 183, 274
  GnuTLS 308
  OpenSSL 307
tls_tempfail_tryclear 274
tls_try_verify_hosts 183

tls_verify_certificates 183, 274
tls_verify_hosts 183
to 255
translate_ip_address 196
transport 197
transport_current_directory 197
transport_filter 233
transport_filter_timeout 235
transport_home_directory 197
trusted_groups 184
trusted_users 184

U

umask 263
unknown_login 184
unknown_username 184
unseen 12, 190, 197, 380
untrusted_set_sender 184
use_bsmtp 247, 264
use_classresources 264
use_crlf 247, 264
use_fcntl_lock 247
use_flock_lock 247
use_lockfile 248
use_mbx_lock 248
use_shell 264
user 198, 235
uucp_from_pattern 185, 377
uucp_from_sender 185, 377

V

verify 198
verify_only 198
verify_recipient 198
verify_sender 198

W

warn_message_file 185
widen_domains 202
write_rejectlog 185

Variables index

$1, $2, etc. *see numerical variables*
$acl_verify_message 119, 219, 320, 324, 325, 342
$address_data 119, 130, 186, 189, 215, 333
$address_file 120, 181, 222, 238
$address_pipe 120, 181, 225, 258
$auth1, *$auth2*, etc 120, 295
$authenticated_id 120, 291, 295, 299, 378, 380
$authenticated_sender 120, 291
$authentication_failed 120
*$bheader_* 101
$body_linecount 120, 431
$body_zerocount 120
$bounce_recipient 120, 390
$bounce_return_size_limit 120, 390
$caller_gid 121, 126
$caller_uid 121, 127, 184
$compile_date 121
$compile_number 121
$demime_errorlevel 121, 359
$demime_reason 121, 359
$dnslist_domain 121, 336
$dnslist_matched 121, 336
$dnslist_text 121, 336
$dnslist_value 121, 336
$domain 121, 126, 153, 156, 176, 212, 229, 236, 272, 277, 278, 279, 280, 318, 332, 371, 374, 388
$domain_data 122, 188, 331
$exim_gid 122
$exim_path 122
$exim_uid 122
$found_extension 122, 359
*$header_* 101
$home 13, 122, 187, 195, 231
$host 122, 207, 211, 212, 234, 266, 271, 273, 274, 293, 300, 311, 389
$host_address 122, 191, 196, 234, 266, 271, 273, 274, 293, 300, 311
$host_data 122, 331
$host_lookup_deferred 123, 131
$host_lookup_failed 123, 130, 160
$inode 123, 241
$interface_address 123
$interface_port 123
$item 101, 104, 106, 114, 123
$ldap_dn 123
$load_average 123
$local_part 13, 123, 126, 156, 187, 191, 217, 229, 236, 238, 258, 277, 278, 279, 280, 318, 326, 371, 374, 394
$local_part_data 124, 192, 331
$local_part_prefix 13, 123, 124, 191
$local_part_suffix 13, 64, 123, 124, 396
$local_scan_data 124, 362
$local_user_gid 13, 124
$local_user_uid 13, 124
$localhost_number 124, 164
$log_inodes 125, 152
$log_space 125, 152
$mailstore_basename 125
$malware_name 125, 353
$max_received_linelength 125
$message_age 125
$message_body 125, 165
$message_body_end 125, 165
$message_body_size 125
$message_exim_id 125
$message_headers 125
$message_headers_raw 125
$message_linecount 126
$message_size 126, 251, 318
$original_domain 126, 229
$original_local_part 126, 187
$originaltor_uid 127
$originator_gid 126
$parent_domain 127
$parent_local_part 127, 187
$pid 127
$pipe_addresses 127, 234, 236, 258, 259, 264
$primary_hostname 127, 168, 175
$qualify_domain 30, 120, 127, 345, 378, 380
$qualify_recipient 127, 225
$rcpt_count 127, 318
$rcpt_defer_count 127
$rcpt_fail_count 127
$received_count 127
$received_for 128
$received_ip_address 128
$received_port 128
$received_protocol 43, 128, 292
$received_time 128
$recipient_data 128
$recipient_verify_failure 128, 342
$recipients 129, 372
$recipients_count 129, 318
$regex_match_string 129
$reply_address 129
$return_path 129, 233
$return_size_limit 129
*$rheader_* 101
$runrc 106, 129
$self_hostname 129, 195
$sender_address 130, 253, 276, 292, 318
$sender_address_data 130, 186, 333
$sender_address_domain 130, 332
$sender_address_local_part 130
$sender_data 130
$sender_fullhost 130, 409
$sender_helo_name 130
$sender_host_address 130, 318, 368
$sender_host_authenticated 130, 292
$sender_host_name 130, 160
$sender_host_port 131
$sender_ident 131, 184
$sender_rcvhost 131, 409
$sender_verify_failure 131, 342
$sending_ip_address 131
$sending_port 132
$smtp_active_hostname 132, 175
$smtp_command 132, 318
$smtp_command_argument 132, 318
$smtp_count_at_connection_start 132
$smtp_notquit_reason 316
$spool_directory 132
$spool_inodes 132, 152

$spool_space 132, 152
$thisaddress 133
$tls_certificate_verified 133
$tls_cipher 133, 149, 266, 292, 310, 311
$tls_peerdn 133, 266, 307, 310, 311
$tod_bsdinbox 133
$tod_epoch 133
$tod_full 133
$tod_log 133, 164
$tod_logfile 133
$tod_zone 133, 164
$tod_zulu 133
$value 100, 103, 106, 133, 209
$version_number 133
$warn_message_delay 134, 391
$warn_message_recipients 134, 391

Concept index

Symbols

.ifdef 51
.include in configuration file 49
.include_if_exists in configuration file 49
@ in a domain list 57, 88
@ in a host list 90
@@ with single-key lookup 96
@[] in a domain list 88
@[] in a host list 90
@mx_any 88
@mx_primary 88
@mx_secondary 88
+caseful 97, 116
+defer_unknown 334
+exclude_unknown 334
+ignore_defer 93
+ignore_unknown 93
+include_defer 93
+include_unknown 93, 334
*@ with single-key lookup 73, 95
/dev/null 218
/etc/aliases 23
/etc/mail/mailer.conf 26
/etc/passwd 72, 187
  multiple reading of 157
/etc/userdbshadow.dat 70

Digits

4*xx* responses
  count of 127
  retry rules for 283
  retrying after 266
  to STARTTLS 274
8-bit characters 28, 146, 168
8BITMIME 146

A

abandoning mail 40, 219
accept ACL verb 319
accept router 199
access control lists (ACLs)
  at start of non-SMTP message 147
  case of local part in 326
  certificate verification 332
  conditions; list of 330
  conditions; processing 321
  customized test 330
  data for message ACL 318
  data for non-message ACL 318
  default configuration 59
  description 314–350
  finding which to use 317
  for non-SMTP messages 147
  format of 319
  indirect 330
  introduction 8
  modifiers; list of 322
  modifiers; processing 321
  nested 330
  on SMTP connection 147
  options for specifying 314
  relay control 349
  return codes 317
  rewriting addresses in 276
  scanning for spam 332
  scanning for viruses 331
  setting up for SMTP commands 147
  testing a DNS list 330, 334
  testing a local part 331
  testing a recipient 332
  testing a recipient domain 331
  testing a sender 332
  testing a sender domain 332
  testing a TLS certificate 332
  testing by regex matching 331, 332
  testing for authentication 330
  testing for encryption 331
  testing the client host 331
  testing, customized 330
  unset options 318
  variables 321
  variables, handling unset 180
  verbs, definition of 319
  verifying header syntax 332
  verifying HELO/EHLO 333
  verifying host reverse lookup 333
  verifying recipient 333
  verifying sender 333
  verifying sender in the header 332
  virus scanning 331
acl ACL condition 330
adding drivers 436
additional groups 191, 232
address
  constructed 381
  copying routing 202, 208
  qualification 168, 376
  rewriting *see rewriting*
  sender 38
  source-routed 167
  testing 34, 189
  verification 35
  without domain 4
address duplicate, discarding 13, 220
address expansion item 108
address list
  @@ lookup type 96
  case forcing 97
  empty item 94
  in a rewriting pattern 278
  in expansion condition 116
  local part starting with ! 95
  lookup for complete address 95
  patterns 94
  regular expression in 95
  split local part and domain 96
address qualification, suppressing 31
address redirection
  broken files 226
  disabling rewriting 226
  domain; preserving 226
  errors 220
  included external list 218

local part without domain 217
non-filter list items 217
one-time expansion 225
redirect router 216
repeated for each delivery attempt 220
to black hole 219
to file 218
to local mailbox 217
to pipe 218
while verifying 216, 347
addresses expansion item 108
admin user 148, 422, 429
definition of 27
alias file
backslash in 217
broken 226
building 27, 31
exception to default 219
in a *redirect* router 216
one-time expansion 225
ownership 225
per-domain default 73
alias for host 92
alternate configuration file 35
"and" expansion condition 118
angle brackets, excess 180
appendfile transport 238–252
appending to a file 248
asterisk
after IP address 384, 405
in address list 96
in domain list 89
in host list 90, 92
in lookup type 89
in search type 74
Athena 7
AUTH
ACL for 147, 314
advertising 149
advertising when encrypted 149
configuration 49, 66
description of 289
in *plaintext* authenticator 295
logging 405
on bounce message 151
on MAIL command 120, 148, 291, 294
testing a server 293
with PAM 117
authenticated ACL condition 330
authentication 289–294
ACL checking 330
advertising 149
bounce message 151
CRAM-MD5 mechanism 299
failure 120
generic options 290
id 120
id, specifying for local message 42
logging 405
LOGIN mechanism 296
Microsoft Secure Password 304
name, specifying for local message 42
NTLM 304
on an Exim client 293
on an Exim server 292
optional in client 271
PLAIN mechanism 295
required by client 271
sender 120
sender, specifying for local message 43
sender; authenticated 291
testing a server 293
authenticators
cram_md5 299–300
cyrus_sasl 301–302
dovecot 303
plaintext 295–298
spa 304–305
autoreply transport 253–255
for system filter 181

B

background delivery 41
backlog of connections 176
backslash in alias file 217
bang paths
not handled by Exim 3
rewriting 281
banner for SMTP 176
base36 9
base62 9, 241
base62 expansion item 108
base62d expansion item 108
base64 encoding
conversion from hex 110
creating authentication test data 293
functions for *local_scan()* use 367
in encrypted password 113
in header lines 101
in *plaintext* authenticator 295
in string expansion 112
batched local delivery 236
batched SMTP input 33, 389
batched SMTP output 388
batched SMTP output example 212
BATV, verifying 348
bcc recipients, verifying none 333
Bcc: header line 46, 378
Berkeley DB library 18
file format 70
BIN_DIRECTORY 23
binary zero
in authentication data 293
in header line 101
in lookup key 70, 71, 417
in message body 120, 125
in *plaintext* authenticator 295
in RFC 2047 decoding 369
bind IP address 271
black hole 219
black list (DNS) 121, 330, 334, 409
body of message
binary zero count 120
definition of 4
expansion variable 125
line count 120
size 125
transporting 230
visible size 165
books about Exim 1

boolean configuration values 52
Bounce Address Tag Validation *see BATV*
bounce message
  copy to other address 156
  customizing 150, 390
  definition of 4
  discarding 161
  failure to deliver 16
  generating 37
  including body 150
  including original 150
  recipient of 16
  redirection details; suppressing 224
  Reply-to: in 156
  sender authentication 151
  size limit 150
  when generated 15
bounce messages
  From: line, specifying 155
broken alias or forward files 226
BSD, DBM library for 17
bug reports 3
Bugzilla 2
build directory 20
build-time options, overriding 21
building alias file 31
building DBM files 417
building Exim 17–23
  architecture type 21
  multiple OS/architectures 17
  operating system type 21
  OS-specific C header files 22
  overriding default settings 21
  pre-building configuration 18
building Eximon 23

C

caching
  callout 346
  callout timeouts 151
  lookup data 75
  named lists 87
caching callout, suppressing 345
callout
  additional parameters for 344
  cache, description of 346
  cache, suppressing 345
  caching timeouts 151
  connection timeout, specifying 344
  defer, action on 344
  full postmaster check 344
  overall timeout, specifying 344
  postmaster; checking 345
  "random" check 345
  sender for recipient check 345
  sender when verifying header 344
  timeout, specifying 344
  verification 343
carriage return 247, 264, 376, 383, 386
case forcing in address lists 97
case forcing in strings 110, 112
case of local parts 13, 97, 187, 326, 381
case sensitivity
  in (n)wildlsearch lookup 71
  in lsearch lookup 71
Cc: header line 46
cdb
  acknowledgment 6
  description of 70
  including support for 22
certificate
  client, location of 273
  references to discussion 312
  revocation list 311
  revocation list for client 273
  revocation list for server 182
  self-signed 312
  server, location of 182
  verification of client 183, 310, 332
  verification of server 274
change log 2
checking access 417
checking disk space 152, 176
CIDR notation 91, 110
cipher
  logging 310, 311
  requiring specific 183, 274
class resources (BSD) 264
Client SMTP Authorization *see CSA*
client, non-queueing 399
command line
  addresses with **-t** 157
  options 27–47
common option syntax 51
concurrent deliveries 228
condition ACL condition 330
configuration file
  alternate 35, 48
  common option syntax 51
  conditional skips 51
  default "walk through" 57–66
  editing 22
  errors in 48
  format of 49
  general description 48
  including other files 49
  leading white space in 49
  macros 50
  main section 141–185
  ownership 48
  retry section 282–288
  trailing white space in 49
configuration for building Exim 18
configuration options
  extracting 31
  hiding value of 32, 52
CONFIGURE_FILE 23, 35, 48
CONFIGURE_GROUP 48
CONFIGURE_OWNER 48
connection backlog 176
constructed address 381
content scanning
  at ACL time 351–360
  for spam 354
  for viruses 351
  MIME checking 359
  MIME parts 356
  per user 234
  specifying at build time 19
  the **malware** condition 351

with regular expressions 358
continue ACL modifier 322
control ACL modifier 322, 326
control of incoming mail 314
copy of bounce message 156
copy of message (**unseen** option) 197
Courier 70
CR character *see carriage return*
cram_md5 authenticator 299–300
CRAM-MD5 authentication mechanism 299
creating directories 238
CRL *see certificate revocation list*
crypt() 113
crypt16() 113
crypteq expansion condition 113
CSA
in *dnsdb* lookup 76
verifying 347
CSA verification 332
current directory for local transport 197, 229
customizing
ACL condition 330
batching condition 236
bounce message 150, 390
"cannot route" message 186
failure message 219
input scan using C function 361
precondition 14
Received: header 171
SMTP banner 176
warning message 185, 391
cycling logs 402, 415
Cygwin 9
Cyrus 6, 117, 118, 262, 265, 267
SASL library 301
cyrus_sasl authenticator 301–302

D

daemon 28, 385
listening IP addresses 137
pid file path 167
process id (pid) 28, 32, 43
restarting 28, 427
starting 137
TCP_NODELAY on sockets 181
daemon startup, retrying 152
Darwin 9
DATA
ACL for 147
ACLs for 314, 315
database
lookups 69–84
updating in ACL 322
Date: header line 378
DBM
building dbm files 417
lookup type 70
DBM libraries
configuration for building 18, 22
discussion of 17
Debian
information sources 1
mailing list for 2
debugging
-bh option 30
-d option 36
-N option 40
from embedded Perl 136
list of selectors 36
suppressing delivery 40
decode ACL condition 330
def expansion condition 114
default
ACLs 59
configuration file "walk through" 57
in single-key lookups 73
retry rule 65
routers 62
transports 65
defer ACL verb 319
defer in system filter 372
defer, fake 326
deferred delivery, forcing 219
delay ACL modifier 323
delay warning, specifying 153
delayed delivery, logging 409
delivery
abandoning further attempts 39
by external agent 264
cancelling all 40
cancelling by address 40
deferral 15
delaying certain domains 160
discarded; logging 406
failure report *see bounce message*
failure; logging 406
fake; logging 406
first 114
forcing attempt 38
forcing deferral 219
forcing failure 219, 425
forcing in queue run 45
from given sender 46
in detail 14
in the background 41
in the foreground 41
log line format 192
manually started – not forced 39
maximum number of 172
parallelism for remote 172
permanent failure 15
problems with 25
procmail 264
retry in remote transports 15
retry mechanism 15
sorting remote 172
suppressing immediate 41
temporary failure 15
to file; forbidding 222
to given domain 45
to pipe; forbidding 224
to single file 250
unprivileged 153
delivery failure, long-term 287
Delivery-date: header line 153, 230, 378
demime ACL condition 330
deny ACL verb 319
design philosophy 8
dialup *see intermittently connected hosts*
directories, multiple 179

directory creation 238, 240, 248, 250
discard ACL verb 320
discarded messages 406
discarding bounce message 161
disk space, checking 152, 176
distribution
  ftp site 3
  public key 3
  signing details 3
dlfunc 99
dmbnz lookup type 70
DNS
  as a lookup type 72, 75
  IPv6 lookup for AAAA records 155
  pre-check of name syntax 154
  qualifying single-component names 201
  resolver options 155, 201, 202
  reverse lookup 131, 159, 433
  "try again" response; overriding 154
DNS list
  data returned from 336
  in ACL 330, 334
  information from merged 338
  IPv6 usage 339
  keyed by domain name 334
  keyed by explicit IP address 334
  logging defer 409
  matching specific returned data 336
  multiple keys for 335
  variables set from 336
DNS resolver, debugging output 37
dnsdb lookup 75
dnslists ACL condition 330
dnslookup router 200–203
doc/ChangeLog 2
doc/NewStuff 2
doc/spec.txt 2
documentation 1
  available formats 3
domain
  ACL checking 331
  definition of 5
  delaying delivery 160
  delivery to 45
  extraction 108
  for qualifying addresses 168
  in redirection; preserving 226
  manually routing 207
  partial; widening 202
  specifying non-immediate delivery 169
  UTF-8 characters in 149
  virtual 395
domain list
  asterisk in 89
  in expansion condition 116
  matching by lookup 89
  matching "ends with" 89
  matching literal domain name 89
  matching local IP interfaces 88
  matching MX pointers to local host 88
  matching primary host name 88
  matching regular expression 89
  patterns for 88
domain literal 88, 148
  default router 62
  recognizing format 58
  routing 204
domainless addresses 4
domains ACL condition 331
dot
  in incoming non-SMTP message 38, 42
  in local part 382
  trailing on domain 180
dovecot authenticator 303
drivers
  adding new 436
  configuration format 55
  definition of 11
  instance definition 11
drop ACL verb 320
dsearch lookup type 70
duplicate addresses 13, 34, 190, 220

E

EACCES 224
EHLO 383, 404
  accepting junk data 159
  ACL for 147, 314, 315
  argument, setting 269
  avoiding use of 270
  forcing reverse lookup 159
  invalid data 386
  underscores in 159
  verifying 333
  verifying, mandatory 160
  verifying, optional 159
empty item in hosts list 90
encrypted ACL condition 331
encrypted strings, comparing 113
encryption
  checking in an ACL 331
  including support for 19
  on SMTP connection 182, 306–313
endpass ACL modifier 324
ENOTDIR 224
envelope sender 27, 30, 38, 163, 184, 188, 232, 247, 264, 373, 377, 394
  rewriting at transport time 277
envelope, definition of 5
Envelope-to: header line 155, 230, 236, 378
environment for local transports 228–229
environment for *pipe* transport 261
environment for pipe transport 259
eq expansion condition 114
eqi expansion condition 114
error
  ignored 407
  in configuration file 48
  in outgoing SMTP 384
  reporting 42
  skipping bad syntax 226
escape characters in quoted strings 53
escape expansion item 108
ESMTP, avoiding use of 270
ETRN
  ACL for 147, 314
  command to be run 176
  logging 409
  processing 387
  serializing 177

value of *$domain* 122
eval expansion item 109
exec failure 261
exicyclog 402, 415
exigrep 415
Exim arguments, logging 409
Exim binary, path name 156
Exim group 156
Exim monitor
acknowledgment 7
description 422–425
window size 423
Exim user 157
exim_checkaccess 417
exim_dbmbuild 417
exim_dumpdb 418
exim_fixdb 419
EXIM_GROUP 48
exim_lock 420
exim_monitor/EDITME 19, 422
exim_tidydb 419
EXIM_USER 48
eximon 422
eximstats 416
exinext 418
exipick 415
exiqgrep 413
exiqsumm 414
exiscan *see content scanning*
exists, expansion condition 114
exiwhat 168, 413
expansion
"and" of conditions 118
arithmetic expression 109
base64 encoding 112
calling Perl from 104
case forcing 110, 112
character translation 108
checking for empty variable 114
checking header line existence 114
combining conditions 118
conditional 103
conditions 112–119
conversion to base 62 108
definition of 53
domain extraction 108
encrypted comparison 113
escape sequences 98
escaping non-printing characters 108
expression evaluation 109
extracting substrings by key 100
extracting substrings by number 100
file existence test 114
first delivery test 114
forall condition 114
forany condition 114
forced failure 99
header insertion 101
hex to base64 110
hmac hashing 102
including literal text 98
inserting an entire file 105
inserting from a socket 105
IP address masking 110
LDAP authentication test 115
list creation 104
local part extraction 110
lookup in 103
MD5 hash 111
negating a condition 112
non-expandable substrings 98
numeric comparison 112
numeric hash 104, 111
of lists 85
of strings 98–134
operators 99, 108
"or" of conditions 118
PAM authentication test 117
pwcheck authentication test 117
queue runner test 117
quoting 111
Radius authentication 118
re-expansion of substring 109
reducing a list to a scalar 106
regular expression comparison 115
RFC 2047 111
RFC 2822 address handling 108
running a command 106
saslauthd authentication test 118
selecting from list by condition 101
SHA-1 hashing 112
statting a file 112, 223
string comparison 114, 115
string length 112
string substitution 107
string truncation 103, 110
substring expansion 112
substring extraction 107
testing 29, 40, 98
textual hash 101, 110
UTF-8 conversion 110
variables 99
variables, list of 119
variables, set from DNS list 336
EXPN
ACL for 147, 314
error text, display of 219
processing 387
router skipping 189
with **verify_only** 198
external local delivery 264
external transports 4
extract
substrings by key 100
substrings by number 100
EXTRALIBS 22

F

fail
in system filter 372
log line; reducing 372
failing delivery
forcing 219
from filter 220
failover *see fallback*
failure of exec 261
fake defer 326
fake rejection 327
fallback
hosts specified on router 189

hosts specified on transport 267, 268
randomized hosts 271
fallover *see fallback*
FAQ 2
fifo (named pipe) 239
file
appending 248
existence test 114
extracting characteristics 112
how a message is held 10
in redirection list 218
inserting into expansion 105
journal 11
locking 244, 248, 249
lookups 69–84, 103
mailbox; checking existing format 241
MBX format 244
requiring for router 193
too many open 164
transport for system filter 181
filter
enabling use of 220
header lines; adding/removing 373
introduction 8
locking out certain features 223
Sieve *see Sieve filter*
specifying in redirection data 217
system filter 180, 371–374
testing 29
transport filter 122, 127, 233, 259, 273, 383
user for processing 198
filtering all mail 371–374
first delivery 114
first_delivery expansion condition 114
fixed point configuration values 52
forcing delivery 38
foreground delivery 41
format
boolean 52
configuration file 49
fixed point 52
group name 53
integer 52
list item in configuration 54
message 31
octal integer 52
of message id 9
spool files 431–435
string 53
time interval 52
user name 53
forward file
broken 226
one-time expansion 225
ownership 225
testing 29
FreeBSD, MTA indirection 26
freeze in system filter 372
freezing messages 39, 372
allowing in filter 221
sending a message when freezing 157
"From" line 27, 30, 31, 38, 162, 185, 240, 244, 250, 262, 377
from_utf8 expansion item 110
From: header line 27, 378
disabling checking of 162
in bounces 155
frozen messages
display 424
forcing delivery 38, 45, 46
forcing in ACL 327
in queue listing 32
logging skipping 410
manual thaw; testing in filter 371
moving 166
spool data 432
thawing 10, 40, 425
timing out 181
fsync(), disabling 154
FTP site 2, 3

G

gdbm DBM library 17
ge expansion condition 114
"gecos" field, parsing 158
gei expansion condition 114
generic options 55
router 186–198
transport 230–235
gid (group id)
caller 121
Exim's own 156
in *queryprogram* router 214
in spool file 431
local delivery 190
of originating user 126
system filter 181, 371
giving up on messages 39
GnuTLS 306
building Exim with 19
specifying parameters for 308
groups
additional 191, 232
name format 53
trusted 184
gt expansion condition 114
gti expansion condition 114

H

hash function
numeric 104, 111
textual 101, 110
header lines
added; visibility of 329
adding 190
adding in an ACL 328
adding in transport 231
adding; in router or transport 380
adding; in system filter 373
character sets 101
decoding 101
in expansion strings 101
listing 40
maximum size of 159
position of added lines 328
removing 190, 231
removing; in router or transport 380
removing; in system filter 373
rewriting 201

rewriting at transport time 277
transporting 231
verifying syntax 332
verifying the sender in 332
header section
definition of 5
maximum size of 158
HELO 383, 404
accepting junk data 159
ACL for 147, 314, 315
argument, setting 269
forcing reverse lookup 159
forcing use of 270
invalid data 386
underscores in 159
verifying 333
HELO verifying
mandatory 160
optional 159
hex2b64 expansion item 110
hiding configuration option values 32, 52
hints database 15
access by remote transport 235
callout cache 346
data expiry 173, 287
DBM files used for 17
ETRN serialization 388
maintenance 418
not overridden by **-Mc** 39
overriding retry hints 38
remembering routing 44, 266
retry keys 194, 232, 383
serializing deliveries to a host 273
use for retrying 286
hmac 102
HOME 260
home directory
for local transport 197, 229
for router 195
HOST 260
host
ACL checking 331
alias for 92
error 384
for RFC 1413 calls 173
limiting SMTP connections from 174
list of; randomized 208, 271
locally unique number for 164
lookup failures 92
lookup failures, permanent 93
lookup failures, temporary 93
maximum number to try 270, 274
name in SMTP responses 175
name of local 168
not logging connections from 161
rejecting connections from 161
reserved 175
serializing connections 273
treated as local 161
unqualified addresses from 171, 173
verifying reverse lookup 333
host list
empty string in 90
lookup of IP address 91
masked IP address 91
matching host name 92, 93
matching IP addresses 90
mixing names and addresses in 94
patterns in 90
regular expression in 92
host name
lookup, failure of 123
lookup, forcing 160
matched in domain list 88
hosts ACL condition 331
HP-UX 158

I

iconv() support 19
id of message 9
ident *see RFC 1413*
if, expansion item 103
ignoring faulty addresses 226
included address list 218
inclusions in configuration file 49
incoming SMTP over TCP/IP 385
incorporated code 6
inetd 34, 174, 385
installing Exim 23
info documentation 24
install script options 24
testing the script 24
what is not installed 24
integer configuration values 52
integer format 52
InterBase
server list 161
InterBase lookup type 72, 82
interface
address, specifying for local message 43
listening 137
logging 409
network 137
intermittently connected hosts 397
IP address
binding 271
discarding 190
for listening 137
masking 91, 110
testing string format 115
translating 196
IP source routing 429
ipliteral router 204
iplookup router 205
iplsearch lookup type 70
IPv6
address scopes 139
addresses in lists 54
disabling 139, 154
DNS black lists 339
DNS lookup for AAAA records 155
including support for 20
IRIX, DBM library for 17
isip expansion condition 115
isip4 expansion condition 115
isip6 expansion condition 115

J

journal file 11

K

keepalive
  on incoming connection 173
  on outgoing connection 272
Kerberos 301

L

lc expansion item 110
LDAP
  authentication 79
  connections 79
  default servers 162
  including support for 22
  lookup type 72
  lookup, more about 77
  protocol version, forcing 162
  query format 78
  quoting 78
  returned data formats 81
  use for authentication 115
  with TLS 78
ldapauth expansion condition 115
le expansion condition 115
lei expansion condition 115
length expansion item 103, 110
length of login name 165
LF character *see linefeed*
LHLO argument setting 269
limit
  bounce message size 150
  hosts; maximum number tried 274
  incoming SMTP connections 174
  message size 165
  message size per transport 232
  messages per SMTP connection 174
  non-mail SMTP commands 174
  number of hosts tried 270
  number of MX tried 270
  number of recipients 172
  on retry interval 173
  open files for lookups 164
  rate of message arrival 177
  retry interval 286
  size of message header section 158
  size of one header line 159
  SMTP connections from one host 174
  SMTP syntax and protocol errors 177
  unknown SMTP commands 177
  user name length 165
limitations of Exim 3
limiting client sending rates 339
line endings 376
line length
  maximum 125
linear search 71
linefeed 247, 264, 376, 383, 386
Linux, DBM library for 17
list
  address list 94
  caching of named 87
  domain list 88
  empty item in 54
  file name in 85
  host list 90
  iterative conditions 114
  local part list 97
  named 86
  named compared with macro 87
  negation 85
  reducing to a scalar 106
  selecting by condition 101
  syntax of in configuration 54
list separator
  changing 54
  newline as 54
listing
  message body 40
  message headers 40
  message in RFC 2922 format 40
  message log 40
  messages on the queue 32
lists of domains; hosts; etc. 85–97
LMTP
  ignoring quota errors 256, 272
  over a pipe 256
  over a socket 256
  over TCP/IP 272, 383
  processing details 383–389
lmtp transport 256
load average 153, 169, 177
  re-evaluating per message 170
local delivery
  definition of 5
  using an external agent 264
local host
  domains treated as 161
  MX pointing to 195, 200
  name of 168
  sending to 196, 267
local message reception 31
local part
  ACL checking 331
  case of 381
  checking in router 192
  definition of 5
  dots in 382
  in retry keys 194
  list 97
  list, in expansion condition 116
  prefix 232, 396
  starting with ! 95, 96
  suffix 232, 396
local SMTP input 34
local transports
  environment for 228–229
  uid and gid 190, 191, 198, 228
local user, checking in router 187
local_part expansion item 110
local_parts ACL condition 331
local_scan() function
  address rewriting; timing of 277
  API description 361
  available Exim functions 366
  available Exim variables 363
  building Exim to use 361
  configuration options 362
  description of 361–370
  memory handling 370
  timeout 163

when all recipients discarded 317
Local/eximon.conf 422
Local/eximon.conf 19, 23
Local/Makefile 18, 21
lock files 25, 241
locking files 241, 242, 244, 248, 249
locking mailboxes 420
log
certificate verification 411
connection rejections 409
cycling local files 402, 415
datestamped files 402
delayed delivery 409
delivery duration 409
delivery line 192, 405
destination 401
distinguished name 310, 311
DNS failure in list 411
dnslist defer 409
dropped connection 409
ETRN commands 409
Exim arguments 409
extracts; grepping for 415
fail command log line 372
file for each message 10
file path for 164
frozen messages; skipped 410
full parentage 409
general description 401–412
header lines for rejection 410
host lookup failure 409
ident timeout 409
incoming interface 409
incoming remote port 409
local files; writing to 402
message log; description of 411
message log; disabling 165
message size on delivery 409
non-MAIL SMTP sessions 411
outgoing remote port 409
process ids in 401, 409
process log 168
queue run 409
queue time 409
reception line 404
recipients 410
retry defer 410
return path 410
rewriting 408
selectors 164, 408
sender on delivery 410
sender reception 410
sender verify failure 410
size rejection 410
smtp confirmation 410
SMTP connections 410
SMTP protocol error 411
SMTP syntax error 411
SMTP transaction; incomplete 411
subject 411
summary of fields 407
syslog; writing to 403
tail of; in monitor 423
timezone for entries 164
TLS cipher 310, 311, 411
TLS peer DN 411
to file 401
to syslog 401
types of 401
unknown SMTP command 411
writing from embedded Perl 136
log_message ACL modifier 324
log_reject_target ACL modifier 324
logging in ACL
immediate 324
specifying which log 324
LOGIN authentication mechanism 296
logwrite ACL modifier 324
lookup
* added to type 73
*@ added to type 73
caching 75
cdb 70
dbm 70
dbm – terminating zero 70
dbmnz 70
default values 73
description of 69
DNS 72
dnsdb 75
dsearch 70
in domain list 89
in expanded string 103
inclusion in binary 22
InterBase 72, 82
iplsearch 70
LDAP 72, 77
lsearch 71
lsearch – colons in keys 71
maximum open files 164
MySQL 72, 82
NIS 71
NIS+ 72, 81
nwildlsearch 71
Oracle 72, 82
partial matching 74
partial matching – changing prefix 74
passwd 72
PostgreSQL 72, 82
query-style types 72
quoting 75
single-key types 70
SQLite 84
sqlite 73
temporary error in 73
types of 69
whoson 73
wildcard 73, 74
wildlsearch 71
loop
caused by **fail** 372
in lookups 96
prevention 171
while file testing 249
while routing 14
while routing, avoidance of 217
lower casing 110, 417
lsearch lookup type 71
lt expansion condition 115
lti expansion condition 115

M

macro
  compared with named list 87
  description of 50
  setting on command line 36
MAIL
  ACL for 147, 314
  logging session without 411
  rewriting argument of 279
  SIZE option 386
mail hub example 212
mail loop prevention 171
mailbox
  locking, blocking and non-blocking 242
  maintenance 420
  MMDF format 240
  multiple 191, 396
  size warning 247
  specifying size of 243
  symbolic link 239, 249
  time of last read 285
maildir format 250
  description of 250
  maildirfolder file 244
  maildirsize file 243, 251
  quota; directories included in 243
  specifying 243
  time of last read 285
maildir++ 251
maildirfolder, creating 244
mailing lists 392
  closed 393
  for Exim users 2
  one-time expansion 225
  re-expansion of 393
  syntax errors in 392
mailq 27
mailstore format 250
  description of 252
  specifying 244
main configuration 141–185
main log 401
maintaining Exim's hints database 418
malware ACL condition 331
manualroute router 207–213
mask expansion item 110
masked IP address 110
match expansion condition 115
match_address expansion condition 116
match_domain expansion condition 116
match_ip expansion condition 116
match_local_part expansion condition 116
maximum *see also limit*
  line length 125
MBX format, specifying 244
md5 expansion item 111
MD5 hash 111, 113
message
  abandoning delivery attempts 39
  adding recipients 38
  age of 125
  changing sender 39
  controlling incoming 314
  copying every 397
  discarded 406
  error 384
  forced failure 372
  format 31
  freezing 372
  frozen 10
  general processing 375–382
  header, definition of 5
  life of 10
  listing body of 40
  listing header lines 40
  listing in RFC 2822 format 40
  listing message log 40
  log file for 10, 411
  manually discarding 40
  manually freezing 39
  queueing by file existence 169
  queueing by load 169
  queueing by message count 175
  queueing by SMTP connection count 175
  queueing certain domains 169
  queueing remote deliveries 170
  queueing unconditionally 169
  reception 9
  size 126
  size in queue listing 32
  size issue for transport filter 273
  size limit 165
  submission 328, 375, 378, 379
  submission, ports for 58
  thawing frozen 10, 40
  transporting body only 230
  transporting headers only 231
message ACL modifier 325, 328
  with **accept** 319
message body
  binary zero count 120
  in expansion 125
  line count 120
  newlines in variables 165
  size 125
  visible size 165
message ids
  details of format 9
  with multiple hosts 164
message logs
  disabling 165
  preserving 168
message sender, constructed by Exim 9
Message-ID: header line 165, 379
Microsoft Secure Password Authentication 6, 304
MIME content scanning 356, 359
  ACL for 148, 314
  returned variables 357
mime_regex ACL condition 331
mixed-case login names 382
MMDF format mailbox 240
monitor *see Exim monitor*
msglog directory 411
multiline responses, suppressing 327
multiple mailboxes 191, 396
multiple SMTP deliveries 39, 41, 44, 267, 270, 398
multiple SMTP deliveries with TLS 311
multiple spool directories 179
MX record
  checking for secondary 200

in *dnsdb* lookup 76
maximum tried 270
pointing to IP address 149
pointing to local host 195, 200
required to exist 201
MySQL
lookup type 72, 82
server list 166

N

Nagle algorithm 181
name
of local host 168
of sender 38
name server for enclosing domain 76
named lists 86
named pipe (fifo) 239
ndbm DBM library 17
negation
in expansion condition 112
in lists 85
network interface 137
new drivers, adding 436
newaliases 27
newline
as list separator 54
in message body variables 165
NFS 194
checking for file existence 222
lock file 241, 248
NIS lookup type 71
including support for 22
NIS, retrying user lookups 157
NIS+ lookup type 72, 81
including support for 22
no_*xxx* *see* entry for xxx
non-immediate delivery 41
non-SMTP messages
ACLs for 147, 314, 315
NTLM authentication 304
NUL *see binary zero*
number of deliveries 172
numeric comparison 112
numerical variables (*$1 $2* etc)
in *manualroute* router 209
numerical variables (*$1 $2* etc) 119
in *cram_md5* authenticator 299
in **errors_copy** 156
in "From " line handling 377
in **gecos_name** 158
in **if** expansion 115
in lookup expansion 104
in *plaintext* authenticator 295
in rewriting rules 278
in *spa* authenticator 304
nwildlsearch lookup type 71

O

one-time aliasing/forwarding expansion 225
open files, too many 164
OpenSSL 306
building Exim with 19
options
appendfile transport 239
autoreply transport 253
command line 27–47
command line; terminating 28
configuration – extracting 31
cram_md5 authenticator (client) 299
cram_md5 authenticator (server) 299
dnslookup router 200
generic – definition of 55
generic; for authenticators 290
generic; for routers 186–198
generic; for transports 230–235
hiding value of 32, 52
iplookup router 205
lmtp transport 256
manualroute router 207
pipe transport 260
plaintext authenticator (client) 297
plaintext authenticator (server) 295
queryprogram router 214
redirect router 220
router – extracting 32
smtp transport 266
spa authenticator (client) 304
spa authenticator (server) 304
transport – extracting 32
"or" expansion condition 118
Oracle
lookup type 72, 82
server list 166
os.h 22
outgoing LMTP over TCP/IP 383
outgoing SMTP over TCP/IP 383
ownership
alias file 225
configuration file 48
forward file 225

P

packet radio 196
PAM authentication 117
pam expansion condition 117
panic log 401
partial matching 74
passwd file *see /etc/passwd*
passwd lookup type 72
PCRE 6, 67
pcretest 67
"percent hack" 167, 349
periodic queue running 45
Perl
calling from Exim 135–136
including support for 22
standard output and error 136
starting the interpreter 44
use in expanded string 104
use of **warn** 136
pid (process id)
in log lines 401, 409
of current process 127
of daemon 28, 32, 43
re-use of 9
pid file, path for 167
pipe
duplicated 220
in redirection list 218

named (fifo) 239
pipe transport 258–265
environment for command 259, 261
failure of exec 261
for system filter 181
logging output 261
path for command 259
permitted commands 260
returned data 258
temporary failure 263
uid for 229
with multiple addresses 236
PIPELINING
avoiding the use of 270
expected errors 177
suppressing advertising 167, 328
PLAIN authentication mechanism 295
plaintext authenticator 295–298
policy control
access control lists 314
address verification 342
by local scan function 361
checking access 417
overview 8
rejection, returning details 178
relay control 349
testing 30
port
465 and 587 58
for daemon 152
for message submission 58
iplookup router 205
logging outgoint remote 409
logging remote 409
receiving TCP/IP 43
sending TCP/IP 272
PostgreSQL lookup type 72, 82
server list 167
preconditions
checking 12
definition of 11
order of processing 13
prefix
for local part, including in envelope 232
for local part, used in router 191
for partial matching 74
preserving domain in redirection 226
primary host name 88
printing characters 168
private options 55
privilege, running without 427
privileged user 429
process id *see pid*
process log path 168
process, querying 413
procmail 264
protocol, specifying for local message 43
prvs expansion item 104
prvscheck expansion item 105
public key for signed distribution 3
pwauthd daemon 6
pwcheck daemon 6, 117
pwcheck expansion condition 117

Q

query-style lookup
definition of 70
list of types 72
queryprogram router 214–215
queue
count of messages on 32
definition of 5
delivering specific messages 45
display in monitor 424
double scanning 44
forcing delivery 45
grepping 413
initial delivery 45
listing messages on 32
local deliveries only 45
menu in monitor 424
routing 44
summary 414
queue runner 14, 15, 27, 28
abandoning 153
definition of 5
description of operation 44
detecting when delivering from 117
for specific recipients 45
for specific senders 46
logging 409
maximum number of 170
processing messages in order 170
starting manually 44
starting periodically 45
queue_running expansion condition 117
queueing incoming messages 41, 169, 170, 175, 328
QUIT, ACL for 148, 314, 316
quota
error testing in retry rule 285
imposed by Exim 245
in maildir delivery 251
maildir; directories included in 243
system 238
warning threshold 247
quote expansion item 111
quote_local_part expansion item 111
quoting
in lookups 75
in pipe command 259
in regular expressions 112
in string expansions 111
lookup-specific 111

R

Radius 118
radius expansion condition 118
randomized host list 208, 271
rate limiting 332
client sending 339
rate limitint
reading data without updating 342
RBL *see DNS list*
RCPT
ACL for 60, 148, 314
maximum number of incoming 172
maximum number of outgoing 272
rate limiting 177

rewriting argument of 279
value of *$message_size* 126
readfile expansion item 105
readsocket expansion item 105
Received: header line 379
counting 171
customizing 171
receiving mail 9
recipient
ACL checking 332
adding 38
adding in local scan 364
error 385
extracting from header lines 46
maximum number 172
removing 40
removing in local scan 364
verifying 333
recipients ACL condition 332
redirect router 216–227
redirection *see address redirection*
References: header line 379
regex ACL condition 332
regular expressions
content scanning with 358
in address list 95
in domain list 89
in host list 92
in retry rules 283
library 6, 67
match in expanded string 115
quoting 112
testing 67
reject log 401
disabling 185
rejection, fake 327
relaying
checking control of 350
control by ACL 349
testing configuration 30
remote delivery, definition of 5
removing messages 40
removing recipients 40
repeated redirection expansion 220
replacing another MTA 25
reporting bugs 3
require ACL verb 320
Resent- header lines 377
with **-t** 46
resolver, debugging output 37
retry
after long-term failure 287
algorithms 286
configuration testing 33
configuration, description of 282–288
default rule 65
description of mechanism 15
fixed intervals 286
increasing intervals 286
intermittently working deliveries 288
interval, maximum 286
limit on interval 173
parameters in rules 286
quota error testing 285
random intervals 286
rules 282
rules; sender-specific 285
specific errors; specifying 284
time not reached 282, 407
timeout of data 287
times 418
return code
for **-bm** 31
for **-bS** 33
for **-bt** 34
for **-bv** 35
for **-oee** 42
for **-oem** 42
for **-oep** 42
for bad configuration 48
from **run** expansion 106, 129
return path *see also envelope sender*
changing in transport 232
created from *Sender:* 380
definition of 5
in submission mode 376
Return-path: header line 233, 379
removing 173
reverse DNS lookup 131, 159, 433
revocation list 311
rewriting
addresses 10, 276–281, 382
at transport time 231, 277
bang paths 281
flags 279
header lines 201
logging 408
patterns 278
replacements 279
rules 277
testing 33, 277
timing of 276
whole addresses 280
RFC 1413 30, 173
logging timeout 409
query timeout 173
RFC 2047 19, 280, 358, 368, 381
binary zero in 369
decoding 111
disabling length check 152
expansion operator 111
rfc2047 expansion item 111
rfc2047d expansion item 111
rmail 27
root privilege 426
running without 427
router
adding header lines 190
carrying on after success 197
case of local parts 187
changing address for errors 188
checking for local user 187
checking senders 196
customized precondition 187
customizing “cannot route” message 186
data attached to address 186
definition of 11
discarding IP addresses 190
fallback hosts 189
for verification 12

forcing verification failure 189
go to after "pass" 193
home directory for 195
IP address translation 196
preconditions, order of processing 13
prefix for local part 191
removing header lines 190
requiring file existence 193
restricting to specific domains 188
restricting to specific local parts 192
result of running 12
running details 12
setting group 190
skipping for EXPN 189
skipping when address testing 186
start at after redirection 193
suffix for local part 192
timeout 193
used only when verifying 198
user for filter processing 198
routers
accept 199
default 62
dnslookup 200–203
ipliteral 204
iplookup 205
manualroute 207–213
queryprogram 214–215
redirect 216–227
routing
by external program 214
loops in 14, 217
whole queue before delivery 44
rsmtp 27
run expansion item 106
run time configuration 48
runq 27
rxquote expansion item 112

S

Samba project 6
saslauthd daemon 118
saslauthd expansion condition 118
scanning *see content scanning*
security
build-time features 426
discussion of 426–430
sender
ACL checking 332
address 38, 377
authenticated 120
changing 39
constructed by Exim 9
definition of 5
gid 126
host address, specifying for local message 42
host name, specifying for local message 43
ident string, specifying for local message 43
name 38
setting by untrusted user 184
source of 34
uid 127
verifying 333
verifying in header 332
sender_domains ACL condition 332
sender_retain submission option 375
Sender: header line 27, 379
disabling addition of 162
retaining from local submission 163
senders ACL condition 332
Sendmail compatibility
-bi option 31
-G option ignored 38
-h option ignored 38
-n option ignored 40
-oA option 41
-om option ignored 43
-oo option ignored 43
-t option 46, 157
-U option ignored 47
8-bit characters 28
calling Exim as *newaliases* 27
command line interface 4
"From" line 31, 377, 378
serializing connections 273
set ACL modifier 325
setuid 426
installing Exim with 23
sg expansion item 107
SHA-1 hash 112, 113
sha2 expansion item 112
shadow transport 233
Sieve filter 8
configuring *appendfile* 238
enabling in default router 64
enabling use of 220
forbidding delivery to a file 222
"keep" facility; disabling 222
not available for system filter 14, 180
relative mailbox path handling 238
specifying in redirection data 217
syntax errors in 227
testing 29
vacation directory 226
value of *$address_file* 120
SIGHUP 28, 427
SIGUSR1 413
simultaneous deliveries 228
single-key lookup
definition of 69
list of types 70
size
of bounce, limit 150
of mailbox 243, 247
of message 32, 126, 273, 405, 409
of message, limit 165, 232
of monitor window 423
SIZE option on MAIL command 383, 386
skipping faulty addresses 226
smart host
example router 211, 392
suppressing queueing 399
SMTP
authentication configuration 289–294
batched incoming 33, 389
batched outgoing 388
batched outgoing; example 212
batching over TCP/IP 384
callout verification 343
command, argument for 132

connection backlog 176
connection, ACL for 314, 315
delaying delivery 41
details policy failures 178
encrypted connection 182
encryption 306–313
error codes 219, 224, 325
errors in outgoing 384
host name in responses 175
incoming call count 175
incoming connection count 174, 175
incoming over TCP/IP 385
input timeout 43, 178
limiting incoming message count 174
limiting non-mail commands 174
limiting syntax and protocol errors 177
limiting unknown commands 177
listener 28
local incoming 388
local input 34
logging confirmation 410
logging connections 410
logging incomplete transactions 411
logging protocol error 411
logging syntax errors 411
multiple deliveries 39, 41, 44, 267, 398
non-mail commands 387
outgoing over TCP/IP 383
output flushing, disabling for callout 327, 344
output flushing, disabling for delay 327
passed connection 39, 41, 44, 267, 384, 398
processing details 383–389
protocol errors 387
rate limiting 177
rewriting malformed addresses 279
SIZE 234, 273
smtps protocol 58, 138, 306
ssmtp protocol 58, 138, 306
synchronization checking 176, 326
syntax errors 387
syntax errors; logging 411
testing incoming 30
unknown command; logging 411
unrecognized commands 387
welcome banner 176
smtp transport 266–275
smtps protocol 58, 138, 306
socket, use of in expansion 105
Solaris
DBM library for 17
flock() support 248
LDAP 77
mail command 38
PAM support 117
stopping Exim on 26
sorting remote deliveries 172
source routing
in email address 167
in IP packets 429
SPA authentication 6
spa authenticator 304–305
spam ACL condition 332
spam scanning 354
returned variables 355
SpamAssassin 354
spool directory
checking space 152, 176
creating 24
definition of 5
file locked 407
files 429
files that hold a message 10
format of files 431–435
input sub-directory 10
path to 179
split 179
"spool file is locked" 410
spool files
editing 431
format of 431–435
sprintf() 430
SQL lookup types 82
sqlite lookup type 73, 84
lock timeout 179
src/EDITME 18
SRV record
enabling use of 200
in *dnsdb* lookup 76
required to exist 201
SSL *see TLS*
ssmtp protocol 58, 138, 306
STARTTLS, ACL for 148, 314
stat expansion item 112
statistics 416
statvfs function 423
"sticky" bit 25, 241
str2b64 expansion item 112
string
case forcing 110, 112
comparison 114, 115
expansion *see expansion*
format of configuration values 53
length in expansion 112
list, definition of 54
quoted 53
testing for IP address 115
stripchart 422
strlen expansion item 112
subject, logging 411
submission fixups, suppressing 328
submission mode 328, 375, 378, 379
substr expansion item 107, 112
substring extraction 107, 112
suffix for local part
including in envelope 232
used in router 192
SUPPORT_TLS 19
symbolic link
to build directory 17
to *exim* binary 25
to mailbox 239, 249
to source files 20
synchronization checking in SMTP 176, 326
syntax of common options 51
syslog 401
duplicate log lines; suppressing 180
facility; setting 180
process name; setting 180
timestamps 180
system aliases file 23

system filter 371–374
  specifying 180
  testing 29
system log 401

T

TCP_NODELAY on listening sockets 181
TCP/IP
  logging incoming remote port 409
  logging ougtoing remote port 409
  setting listening interfaces 43, 137
  setting listening ports 43, 137, 152
  setting outgoing port 272
tcpwrappers, building Exim to support 20
tdb DBM library 18
terminology definitions 4
testing
  addresses 34, 189
  filter file 29
  forward file 29
  incoming SMTP 30
  installation 24
  regular expressions 67
  relay control 30
  retry configuration 33
  rewriting 33, 277
  string expansion 29, 40, 98
  system filter 29
  variables in drivers 188, 230
thawing messages 40, 149, 425
time interval
  decoding 112
  formatting 112
  specifying in configuration 52
time_eval expansion item 112
time_interval expansion item 112
timeout
  for *local_scan()* function 163
  for non-SMTP input 43, 171
  for RFC 1413 call 173
  for SMTP input 43, 178
  frozen messages 181
  mailbox locking 242
  of retry data 287
  of router 193
timezone, setting 182
TLS
  advertising 182
  automatic start 47
  avoiding for certain hosts 270
  broken clients 183
  client certificate revocation list 273
  client certificate verification 183, 310, 332
  client certificate, location of 273
  client private key, location of 274
  configuring an Exim client 311
  configuring an Exim server 309
  D-H parameters for server 182
  esmtp state; remembering 183
  including support for TLS 19
  logging cipher 411
  logging peer DN 411
  multiple message deliveries 270, 311
  on SMTP connection 306
  OpenSSL *vs* GnuTLS 306
  passing connection 270
  requiring for certain servers 271
  requiring specific ciphers 183, 274
  requiring specific ciphers (OpenSSL) 307
  revoked certificates 311
  server certificate revocation list 182
  server certificate verification 274
  server certificate; location of 182
  server private key; location of 183
  specifying ciphers (GnuTLS) 308
  specifying key exchange methods (GnuTLS) 308
  specifying MAC algorithms (GnuTLS) 308
  specifying protocols (GnuTLS) 308
  use without STARTTLS 47
tmail 262
To: header line 46
too many open files 164
top bit *see 8-bit characters*
tr expansion item 108
trailing dot on domain 180
training courses 3
transport
  body only 230
  current directory for 230
  definition of 11
  external 4
  filter 122, 127, 233, 259, 273, 383
  filter, timeout 235
  generic options for 230–235
  group; additional 232
  group; specifying 231
  header lines only 231
  header lines; adding 231
  header lines; removing 231
  header lines; rewriting 231
  home directory for 231
  local 190, 191, 198
  local; address batching in 236
  local; current directory for 229
  local; environment for 228–229
  local; home directory for 229
  local; uid and gid 228
  message size; limiting 232
  return path; changing 232
  shadow 233
  user, specifying 235
transports
  appendfile 238–252
  autoreply 253–255
  default 65
  lmtp 256
  pipe 258–265
  smtp 266–275
Tru64-Unix build-time settings 21
trusted groups 184
trusted users 38, 184, 429
  definition of 27

U

uc expansion item 112
uid (user id)
  caller 121
  Exim's own 157
  for *queryprogram* 214
  in spool file 431

local delivery 198, 235, 262
of originating user 127
system filter 181, 371
unknown caller 184
underscore in EHLO/HELO 159
unfreezing messages 40, 149, 425
Unicode 110
unknown host name 92, 93
unprivileged delivery 153
unprivileged running 427
unqualified addresses 171, 173, 376
untrusted user setting sender 184
upgrading Exim 26
upper casing 112
USE_DB 18, 417
USE_GNUTLS 19
USE_TCP_WRAPPERS 20
user
admin 429
admin definition of 27
trusted 38, 184, 429
trusted definition of 27
untrusted setting sender 184
user name
format of 53
maximum length 165
UTF-8
conversion from 110
in domain name 149
utilities 413–421
UUCP
example of router for 213
"From" line 31, 162, 185, 377

V

vacation processing 397
Variable Envelope Return Paths 394
variables *see expansion, variables*
verify ACL condition 332, 333
verifying
EHLO 333
header syntax 332
HELO 333
not blind 333
recipient 333
redirection while 347
sender 333
sender in header 332
suppressing error details 346
verifying address
by callout 343
differentiating failures 342
options for 342
overview 12
using **-bv** 35
VERP 394
version number of Exim 34
virtual domains 395
virus scanners
clamd 352
command line interface 352
DrWeb 352
F-Secure 352
Kaspersky 352, 353
mksd 353
Sophos and Sophie 353
virus scanning 351
VRFY
ACL for 148, 314
error text, display of 219
processing 387

W

warn ACL verb 320, 329
log when skipping 408
warning of delay 153
customizing the message 185, 391
web site 2
welcome banner for SMTP 176
white space
in configuration file 49
in header lines 101
in lsearch key 71
whoson lookup type 73
wiki 2
wildcard lookups 73, 74
wildlsearch lookup type 71
window size 423

X

X-Failed-Recipients: header line 15
X-windows 7, 422
X11 libraries, location of 22

Z

zero, binary *see binary zero*